(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number : 0 622 741 A2

(12) EUROPEAN PATENT APPLICATION

(21) Application number : 94302323.4

(22) Date of filing : 30.03.94

(51) Int. Cl.[5] : G06F 15/332, H04N 7/13, G06F 15/64

The application is published incomplete as filed (Article 93 (2) EPC). The point in the description or the claim(s) at which the omission obviously occurs has been left blank.

(30) Priority : 30.03.93 US 40301
30.07.93 US 100747
01.10.93 US 130571

(43) Date of publication of application :
02.11.94 Bulletin 94/44

(84) Designated Contracting States :
AT BE CH DE DK ES FR GB GR IE IT LI LU MC NL PT SE

(71) Applicant : KLICS, Ltd.
P.P. Box 570,
No.1, Le Couteur Court,
Mulcaster Street,
St Helier
Jersey JE4 8X2, Channel Islands (GB)

(72) Inventor : Knowles, Gregory P.
Calle Menorca 18-2-B
E-07011 Palma (ES)

(74) Representative : Jones, Ian
W.P. THOMSON & CO.
Celcon House
289-293 High Holborn
London WC1V 7HU (GB)

(54) Device and method for data compression/decompression.

(57) An apparatus produces an encoded and compressed digital data stream from an original input digital data stream using a forward discrete wavelet transform and a tree encoding method. The input digital data stream may be a stream of video image data values in digital form. The apparatus is also capable of producing a decoded and decompressed digital data stream closely resembling the originally input digital data stream from an encoded and compressed digital data stream using a corresponding tree decoding method and a corresponding inverse discrete wavelet transform. A dual convolver is disclosed which performs both boundary and nonboundary filtering for forward transform discrete wavelet processing and which also performs filtering of corresponding inverse transform discrete wavelet processes. A portion of the dual convolver is also usable to filter an incoming stream of digital video image data values before forward discrete wavelet processing. Methods and structures for generating the addresses to read/write data values from/to memory as well as for reducing the total amount of memory necessary to store data values are also disclosed.

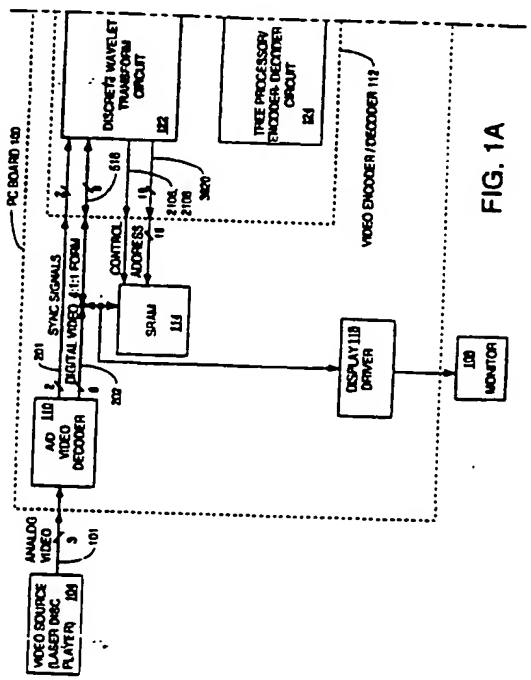

FIG. 1A

EP 0 622 741 A2

CROSS REFERENCE TO PAPER APPENDICES

Appendix A, which is a part of the present disclosure, is a paper appendix of 6 pages. Appendix A is a description of a CONTROL_ENABLE block contained in the tree processor/encoder-decoder portion of a video encoder/decoder integrated circuit chip, written in the VHDL hardware description language.

Appendix B, which is a part of the present disclosure, is a paper appendix of 10 pages. Appendix B is a description of a MODE_CONTROL block contained in the tree processor/encoder-decoder portion of a video encoder/decoder integrated circuit chip, written in the VHDL hardware description language.

Appendix C, which is a part of the present disclosure, is a paper appendix of 11 pages. Appendix C is a description of a CONTROL_COUNTER block contained in the tree processor/encoder-decoder portion of a video encoder/decoder integrated circuit chip, written in the VHDL hardware description language.

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever. The VHDL hardware description language of Appendices A, B and C is an international standard, IEEE Standard 1076-1987, and is described in the "IEEE Standard VHDL Language Reference Manual".

Appendix D, which is a part of the present disclosure, is a paper appendix of 181 pages. Appendix D is a description of one embodiment of a video encoder/decoder integrated circuit chip in the VHDL hardware description language. The VHDL hardware description language of Appendix D is an international standard, IEEE Standard 1076-1987, and is described in the "IEEE Standard VHDL Language Reference Manual". The "IEEE Standard VHDL Language Reference Manual" can be obtained from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoese Lane, Piscataway, New Jersey 08855, telephone 1-800-678-4333.

DESCRIPTION

This invention relates to a method and apparatus for compressing, decompressing, transmitting, and/or storing digitally encoded data. In particular, this invention relates to the compression and decompression of digital video image data.

An apparatus produces an encoded/compressed digital data stream from an original input digital data stream using a discrete wavelet transform and a tree encoding method. The apparatus is also capable of producing a decoded/decompressed digital data stream closely resembling the originally input digital data stream from an encoded/compressed digital data stream using a corresponding tree decoding method and a corresponding inverse discrete wavelet transform.

The apparatus comprises a discrete wavelet transform circuit which is capable of being configured to perform either a discrete wavelet transform or a corresponding inverse discrete wavelet transform. The discrete wavelet transform circuit comprises an address generator which generates the appropriate addresses to access data values stored in memory. Methods and structures for reducing the total amount of memory necessary to store data values and for taking advantage of various types of memory devices including dynamic random access memory (DRAM) devices are disclosed. A convolver circuit of the discrete wavelet transform circuit performs both boundary and non-boundary filtering for the forward discrete wavelet transform and performs start, odd, even and end reconstruction filtering for the inverse discrete wavelet transform. The convolver may serve the dual functions of 1) reducing the number of image data values before subsequent forward discrete wavelet transforming, and 2) operating on the reduced number of image data values to perform the forward discrete wavelet transform.

The apparatus also comprises a tree processor/ encoder-decoder circuit which is configurable in an encoder mode or in a decoder mode. In the encoder mode, the tree processor/encoder-decoder circuit generates addresses to traverse trees of data values of a sub-band decomposition, generates tokens, and quantizes and Huffman encodes selected transformed data values stored in memory. In the decoder mode, the tree processor/decoder-encoder circuit receives Huffman encoded data values and tokens, Huffman decodes and inverse quantizes the encoded data values, recreates trees of transformed data values from the tokens and data values, and stores the recreated trees of data values in memory.

The apparatus is useful in, but not limited to, the fields of video data storage, video data transmission, television, video telephony, computer networking, and other fields of digital electronics in which efficient storage and/or transmission and/or retrieval of digitally encoded data is needed. The apparatus facilitates the efficient and inexpensive compression and storage of video and/or audio on compact laser discs (commonly known as CDs) as well as the efficient and inexpensive storage of video and/or audio on digital video tapes (commonly known as VCR or "video cassette recorder" tapes). Similarly, the invention facilitates the efficient

and inexpensive retrieval and decompression of video and/or audio from digital data storage media including CDs and VCR tapes.

The invention is further described below, by way of example, with reference to the accompanying drawings, in which:

Figure 1 is a block diagram of an expansion printed circuit board which is insertable into a card slot of a personal computer.

Figure 2 is a block diagram of an embodiment of the analog/digital video decoder chip depicted in Figure 1.

Figures 3A-C illustrate a 4:1:1 luminance-chrominance-chrominance format (Y:U:V) used by the expansion board of Figure 1.

Figure 4 is an illustration of a timeline of the output values output from the analog/digital video decoder chip of Figures 1 and 2.

Figure 5 is a block diagram of the discrete wavelet transform circuit of the video encoder/decoder chip of Figure 1.

Figure 6 is a block diagram of the row convolver block of Figure 5.

Figure 7 is a block diagram of the column convolver block of Figure 5.

Figure 8 is a block diagram of the wavelet transform multiplier circuit blocks of Figures 6 and 7.

Figure 9 is a block diagram of the row wavelet transform circuit block of Figure 6.

Figure 10 is a diagram illustrating control signals which control the row convolver of Figure 5 and signals output by the row convolver of Figure 5 during a forward octave 0 transform.

Figure 11 is a diagram showing data flow in the row convolver of Figure 5 during a forward octave 0 transform.

Figure 12 is a diagram illustrating data values output by the row convolver of Figure 5 during the forward octave 0 transform.

Figure 13 is a block diagram of the column wavelet transform circuit block of Figure 7.

Figure 14 is a diagram illustrating control signals which control the column convolver of Figure 5 and signals output by the column convolver of Figure 5 during a forward octave 0 transform.

Figure 15 is a diagram showing data flow in the column convolver of Figure 5 during a forward octave 0 transform.

Figure 16 is a diagram illustrating data values present in memory unit 116 of Figure 1 after operation of the column convolver of Figure 5 during the forward octave 0 transform.

Figure 17 is a diagram showing control signals controlling the row convolver of Figure 5 and signals output by the row convolver of Figure 5 during a forward octave 1 transform.

Figure 18 is a diagram showing data flow in the row convolver of Figure 5 during a forward octave 1 transform.

Figure 19 is a diagram showing control signals controlling the column convolver of Figure 5 and signals output by the column convolver of Figure 5 during a forward octave 1 transform.

Figure 20 is a diagram showing data flow in the column convolver of Figure 5 during a forward octave 1 transform.

Figure 21 is a block diagram of one embodiment of the control block 506 of the discrete wavelet transform circuit of Figure 5.

Figure 22 is a diagram showing control signals controlling the column convolver of Figure 5 and signals output by the column convolver of Figure 5 during an inverse octave 1 transform.

Figure 23 is a diagram showing data flow in the column convolver of Figure 5 during a forward octave 1 transform.

Figure 24 is a diagram showing control signals controlling the row convolver of Figure 5 and signals output by the row convolver of Figure 5 during an inverse octave 1 transform.

Figure 25 is a diagram showing data flow in the row convolver of Figure 5 during an inverse octave 1 transform.

Figure 26 is a diagram showing control signals controlling the column convolver of Figure 5 and signals output by the column convolver of Figure 5 during an inverse octave 0 transform.

Figure 27 is a diagram showing data flow in the column convolver of Figure 5 during an inverse octave 0 transform.

Figure 28 is a diagram showing control signals controlling the row convolver of Figure 5 and signals output by the row convolver of Figure 5 during an inverse octave 0 transform.

Figure 29 is a diagram showing data flow in the row convolver of Figure 5 during an inverse octave 0 transform.

Figure 30 is a block diagram of the DWT address generator block of the discrete wavelet transform circuit

of Figure 5.

Figure 31 is a block diagram of the tree processor/encoder-decoder circuit 124 of Figure 1, simplified to illustrate an encoder mode.

Figure 32 is a block diagram of the tree processor/encoder-decoder circuit 124 of Figure 1, simplified to illustrate a decoder mode.

Figure 33 is a block diagram of the decide circuit block 3112 of the tree processor/encoder-decoder of Figures 31-32.

Figure 34 is a block diagram of the tree processor address generator TP_ADDR_GEN block 3114 of the tree processor/encoder-decoder of Figures 31-32.

Figure 35 illustrates the state table for the CONTROL_ENABLE block 3420 of the tree processor address generator of Figure 34.

Figure 36 is a graphical illustration of the tree decomposition process, illustrating the states and corresponding octaves of Figure 35.

Figure 37 is a block diagram of the quantizer circuit block 3116 of the tree processor/encoder-decoder of Figures 31-32.

Figure 38 is a block diagram of the buffer block 3122 of the tree processor/encoder-decoder of Figures 31-32.

Figure 39 is a diagram of the buffer block 3122 of Figure 38 which has been simplified to illustrate buffer block 3122 operation in the encoder mode.

Figure 40 illustrates the output of barrel shifter 3912 of buffer block 3122 when buffer block 3122 is in the encoder mode as in Figure 39.

Figure 41 is a diagram of the buffer block 3122 of Figure 38 which has been simplified to illustrate buffer block 3122 operation in the decoder mode.

Figure 42 illustrates a pipelined encoding-decoding scheme used by the tree processor/encoder-decoder 124 of Figures 31 and 32.

Figure 43 is a block diagram of another embodiment in accordance with the present invention in which the Y:U:V input is in a 4:2:2 format.

Figure 44 illustrates a sequence in which luminance data values are read from and written to the new portion of memory unit 116 of the PC board 100 in a first embodiment in accordance with the invention in which memory unit 116 is realized as a static random access memory (SRAM).

Figure 45 illustrates a sequence in which luminance data values are read from and written to the new portion of memory unit 116 of the PC board 100 in a second embodiment in accordance with the present invention in which memory unit 116 is realized as a dynamic random access memory (DRAM).

Figure 46 illustrates a third embodiment in accordance with the present invention in which memory unit 116 of the PC board 100 is realized as a dynamic random access memory and in which a series of static random access memories are used as cache buffers between tree processor/encoder-decoder 124 and memory unit 116.

Figure 47 illustrates a time line of the sequence of operations of the circuit illustrated in Figure 46.

Figure 1 illustrates a printed circuit expansion board 100 which is insertable into a card slot of a personal computer. Printed circuit board 100 may be used to demonstrate features in accordance with various aspects of the present invention. Printed circuit board 100 receives an analog video signal 101 from an external video source 104 (such as a CD player), converts information in the analog video signal into data in digital form, transforms and compresses the data, and outputs compressed data onto a computer data bus 106 (such as an ISA/NUBUS parallel bus of an IBM PC or IBM PC compatible personal computer). While performing this compression function, the board 100 can also output a video signal which is retrievable from the compressed data. This video signal can be displayed on an external monitor 108. This allows the user to check visually the quality of images which will be retrievable later from the compressed data while the compressed data is being generated. Board 100 can also read previously compressed video data from data bus 106 of the personal computer, decompress and inverse-transform that data into an analog video signal, and output this analog video signal to the external monitor 108 for display.

Board 100 comprises an analog-to-digital video decoder 110, a video encoder/decoder integrated circuit chip 112, two static random access memory (SRAM) memory units 114 and 116, a display driver 118, and a first-in-first-out memory 120. Analog-to-digital (A/D) video decoder 110 converts incoming analog video signal 101 into a digital format. Video encoder/decoder chip 112 receives the video signal in the digital format and performs a discrete wavelet transform (DWT) function, and then a tree processing function, and then a Huffman encoding function to produce a corresponding compressed digital data stream. Memory unit 116 stores "new" and "old" DWT-transformed video frames.

Video encoder/decoder chip 112 comprises a discrete wavelet transform circuit 122 and a tree proces-

sor/ encoder-decoder circuit 124. The discrete wavelet transform circuit 122 performs either a forward discrete wavelet transformation or an inverse discrete wavelet transformation, depending on whether the chip 112 is configured to compress video data or to decompress compressed video data. Similarly, the tree processor/encoder-decoder circuit 124 either encodes wavelet-transformed images into a compressed data stream or decodes a compressed data stream into decompressed images in wavelet transform form, depending on whether the chip 112 is configured to compress or to decompress video data. Video encoder/decoder chip 112 is also coupled to computer bus 106 via a download register bus 128 so that the discrete wavelet transform circuit 122 and the tree processor/encoder-decoder circuit 124 can receive control values (such as a value indicative of image size) from ISA bus 106. The control values are used to control the transformation, tree processing, and encoding/decoding operations. FIFO buffer 120 buffers data flow between the video encoder/decoder chip 112 and the data bus 106. Memory unit 114 stores a video frame in uncompressed digital video format. Display driver chip 118 converts digital video data from either decoder 110 or from memory unit 114 into an analog video signal which can be displayed on external monitor 108.

Figure 2 is a block diagram of analog/digital video decoder 110. Analog/digital video decoder 110 converts the analog video input signal 101 into one 8-bit digital image data output signal 202 and two digital video SYNC output signals 201. The 8-bit digital image output signal 202 contains the pixel luminance values, Y, time multiplexed with the pixel chrominance values, U and V. The video SYNC output signals 201 comprise a horizontal synchronization signal and a vertical synchronization signal.

Figures 3A-C illustrate a 4:1:1 luminance-chrominance-chrominance format (Y:U:V) used by board 100. Because the human eye is less sensitive to chrominance variations than to luminance variations, chrominance values are subsampled such that each pixel shares an 8-bit chrominance value U and an 8-bit chrominance value V with three of its neighboring pixels. The four pixels in the upper-left hand corner of the image, for example, are represented by $\{Y_{00}, U_{00}, V_{00}\}$, $\{Y_{01}, U_{00}, V_{00}\}$, $\{Y_{10}, U_{00}, V_{00}\}$, and $\{Y_{11}, U_{00}, V_{00}\}$. The next four pixels to the right are represented by $\{Y_{02}, U_{01}, V_{01}, \{Y_{03}; U_{01}, V_{01}\}, \{Y_{12}, U_{01}, V_{01}\}$, and $\{Y_{13}, U_{01}, V_{01}\}$. A/D video decoder 110 serially outputs all the 8-bit Y-luminance values of a frame, followed by all the 8-bit U-chrominance values of the frame, followed by all the 8-bit V-chrominance values of the frame. The Y, U and V values for a frame are output every 1/30 of a second. A/D video decoder 110 outputs values in raster-scan format so that a row of pixel values Y_{00}, Y_{01}, Y_{02} ... is output followed by a second row of pixel values Y_{10}, Y_{11}, Y_{12} ... and so forth until all the values of the frame of Figure 3A are output. The values of Figure 3B are then output row by row and then the values of Figure 3C are output row by row. In this 4:1:1 format, each of the U and V components of the image contains one quarter of the number of data values contained in the Y component.

Figure 4 is a diagram of a timeline of the output of A/D video decoder 110. The bit rate of the decoder output is equal to 30 frames/sec x 12 bits/pixel. For a 640 x 400 pixel image, for example, the data rate is approximately 110×10^6 bits/second. A/D video decoder 110 also detects the horizontal and vertical synchronization signals in the incoming analog video input signal 102 and produces corresponding digital video SYNC output signals 201 to the video encoder/decoder chip 112.

The video encoder/decoder integrated circuit chip 112 has two modes of operation. It can either transform and compress ("encode") a video data stream into a compressed data stream or it can inverse transform and decompress ("decode") a compressed data stream into a video data stream. In the compression mode, the digital image data 202 and the synchronization signals 201 are passed from the A/D video decoder 110 to the discrete wavelet transform circuit 122 inside the video encoder/decoder chip 112. The discrete wavelet transform circuit 122 performs a forward discrete wavelet transform operation on the image data and stores the resulting wavelet-transformed image data in the "new" portion of memory unit 116. At various times during this forward transform operation, the "new" portion of memory unit 116 stores intermediate wavelet transform results, such that certain of the memory locations of memory unit 116 are read and overwritten a number of times. The number of times the memory locations are overwritten corresponds to the number of octaves in the wavelet transform. After the image data has been converted into a sub-band decomposition of wavelet-transformed image data, the tree processor/ encoder-decoder circuit 124 of encoder/decoder chip 112 reads wavelet-transformed image data of the sub-band decomposition from the "new" portion of memory 116, processes it, and outputs onto lines 130 a compressed ("encoded") digital data stream to FIFO buffer 120. During this tree processing and encoding operation, the tree processor/encoder-decoder circuit 124 also generates a quantized version of the encoded first frame and stores that quantized version in the "old" portion of memory unit 116. The quantized version of the encoded first frame is used as a reference when a second frame of wavelet-transformed image data from the "new" portion of memory unit 116 is subsequently encoded and output to bus 106. While the second frame is encoded and output to bus 106, a quantized version of the encoded second frame is written to the "old" portion of memory unit 116. Similarly, the quantized version of the encoded second frame in the "old" portion of memory unit 116 is later used as a reference for encoding a third frame of image data.

In the decompression mode, compressed ("encoded") data is written into FIFO 120 from data bus 106 and

is read from FIFO 120 into tree processor/encoder-decoder circuit 124 of the video encoder/decoder chip 112. The tree processor/encoder-decoder circuit 124 decodes the compressed data into decompressed wavelet-transformed image data and then stores the decompressed wavelet-transformed image data into the "old" portion of memory unit 116. During this operation, the "new" portion of memory unit 116 is not used. Rather, the tree processor/encoder-decoder circuit 124 reads the previous frame stored in the "old" portion of memory unit 116 and modifies it with information from the data stream received from FIFO 120 in order to generate the next frame. The next frame is written over the previous frame in the same "old" portion of the memory unit 116. Once the decoded wavelet-transformed data of a frame of image data is present in the "old" portion of memory unit 116, the discrete wavelet transform circuit 122 accesses memory unit 116 and performs an inverse discrete wavelet transform operation on the frame of image data. For each successive octave of the inverse transform, certain of the memory locations in the "old" portion of memory unit 116 are read and overwritten. The number of times the locations are overwritten corresponds to the number of octaves in the wavelet transform. On the final octave of the inverse transform which converts the image data from octave-0 transform domain into standard image domain, the discrete wavelet transform circuit 122 writes the resulting decompressed and inverse-transformed image data into memory unit 114. The decompressed and inverse-transformed image data may also be output to the video display driver 118 and displayed on monitor 108.

Figure 5 is a block diagram of the discrete wavelet transform circuit 122 of video encoder/decoder chip 112. The discrete wavelet transform circuit 122 shown enclosed by a dashed line comprises a row convolver block CONV_ROW 502, a column convolver block CONV_COL 504, a control block 506, a DWT address generator block 508, a REGISTERS block 536, and three multiplexers, mux1 510, mux2 512, and mux3 514. In order to transform a frame of digital video image data received from A/D video decoder 110 into the wavelet transform domain, a forward two dimensional discrete wavelet transform is performed. Similarly, in order to return the wavelet transform digital data values of the frame into a digital video output suitable for displaying on a monitor such as 108, an inverse two dimensional discrete wavelet transform is performed. In the presently described embodiment of the present invention, four coefficient quasi-Daubechies digital filters are used as set forth in the copending Patent Cooperation Treaty (PCT) application filed March 30, 1994 entitled "Data Compression and Decompression".

The discrete wavelet transform circuit 122 shown in Figure 5 performs a forward discrete wavelet transform as follows. First, a stream of 8-bit digital video image data values is supplied, one value at a time, to the discrete wavelet transform circuit 122 via eight leads 516. The digital video image data values are coupled through multiplexer mux1 510 to the input leads 518 of the row convolver CONV_ROW block 502. The output leads 520 of CONV_ROW block 502 are coupled through multiplexer mux2 512 to input leads 522 of the CONV_COL block 504. The output leads 524 of CONV_COL 504 block are coupled to data leads 526 of memory unit 116 through multiplexer mux3 so that the data values output from CONV_COL block 504 can be written to the "new" portion of frame memory unit 116. The writing of the "new" portion of memory unit 116 completes the first pass, or octave, of the forward wavelet transform. To perform the next pass, or octave, of the forward wavelet transform, low pass component data values of the octave 0 transformed data values are read from memory unit 116 and are supplied to input leads 518 of CONV_ROW block 502 via input leads 526, lines 528 and multiplexer mux1 510. The flow of data proceeds through row convolver CONV_ROW block 502 and through column convolver CONV_COL block 504 with the data output from CONV_COL block 504 again being written into memory unit 116 through multiplexer mux3 514 and leads 526. Control block 506 provides control signals to mux1 510, mux2 512, mux3 514, CONV_ROW block 502, CONV_COL block 504, DWT address generator block 508, and memory unit 116 during this process. This process is repeated for each successive octave of the forward transform. The data values read from memory unit 116 for the next octave of the transform are the low pass values written to the memory unit 116 on the previous octave of the transform.

The operations performed to carry out the inverse discrete wavelet transform proceed in an order substantially opposite the operations performed to carry out the forward discrete wavelet transform. The frame of image data begins in the transformed state in memory unit 116. For example, if the highest octave in the forward transform (OCT) is octave 1, then transformed data values are read from memory unit 116 and are supplied to the input leads 522 of the CONV_COL block 504 via leads 526, lines 528 and multiplexer mux2 512. The data values output from CONV_COL block 504 are then supplied to the input leads 518 of CONV_ROW block 502 via lines 525 and multiplexer mux1 510. The data values output from CONV_ROW block 502 and present on output leads 520 are written into memory unit 116 via lines 532, multiplexer mux3 514 and leads 526. The next octave, octave 0, of the inverse transform proceeds in similar fashion except that the data values output by CONV_ROW block 502 are the fully inverse-transformed video data which are sent to memory unit 114 via lines 516 rather than to memory unit 116. Control block 506 provides control signals to multiplexer mux1 510, multiplexer mux2 512, multiplexer mux3 514, CONV_ROW block 502, CONV_COL block 504, DWT address generator block 508, memory unit 116, and memory unit 114 during this process.

In both forward wavelet transform and inverse wavelet transform operations, the control block 506 is timed by the external video sync signals 201 received from A/D video decoder 110. Control block 506 uses these sync signals as well as register input values ximage, yimage, and direction to generate the appropriate control signals mentioned above. Control block 506 is coupled to: multiplexer mux1 510 via control leads 550, multiplexer mux2 512 via control leads 552, multiplexer mux3 via control leads 554, CONV_ROW block 502 via control leads 546, CONV_COL block 504 via control leads 548, DWT address generator block 508 via control leads 534, 544, and 556, memory unit 116 via control leads 2108, and memory unit 114 via control leads 2106.

As shown in Figure 5, multiplexer mux1 510 couples one of the following three sets of input signals to input leads 518 of CONV_ROW block 502, depending on the value of control signals on leads 550 supplied from CONTROL block 506: digital video input data values received on lines 516 from A/D video decoder 110, data values from memory unit 116 or data values from multiplexer mux3 514 received on lines 528, or data values from CONV_COL block 504 received on lines 525. Multiplexer mux2 512 couples either the data values being output from row convolver CONV_ROW block 502 or the data values being output from multiplexer mux3 514 received on lines 528 to input leads 522 of CONV_COL block 504, depending on the value of control signals on lead 552 generated by CONTROL block 506. Multiplexer mux3 514 passes either the data values being output from CONV_ROW 502 received on lines 532 or the data values being output from CONV_COL 504 onto lines 523 and leads 526, depending on control signals generated by CONTROL block 506. Blocks CONV_ROW 502, CONV_COL 504, CONTROL 506, DWT address generator 508, and REGISTERS 536 of Figure 5 are described below in detail in connection with a forward transformation of a matrix of digital image data values. Lines 516, 532, 528 and 525 as well as input and output leads 518, 520, 522, 524 and 526 are each sixteen bit parallel lines and leads.

Figure 6 is a block diagram of the row convolver CONV_ROW block 502. Figure 7 is a block diagram of the column convolver CONV_COL block 504. Figure 21 is a block diagram of the CONTROL block 506 of Figure 5. Figure 30 is a block diagram of the DWT address generator block 508 of Figure 5.

As illustrated in Figure 6, CONV_ROW block 502 comprises a wavelet transform multiplier circuit 602, a row wavelet transform circuit 604, a delay element 606, a multiplexer MUX 608, and a variable shift register 610. To perform a forward discrete wavelet transform, digital video values are supplied one-by-one to the discrete wavelet transform circuit 122 of the video encoder/decoder chip 112 illustrated in Figure 1. In one embodiment in accordance with the present invention, the digital video values are in the form of a stream of values comprising 8-bit Y (luminance) values, followed by 8-bit U (chrominance) values, followed by 8-bit V (chrominance) values. The digital video data values are input in "raster scan" form. For clarity and ease of explanation, a forward discrete wavelet transform of an eight-by-eight matrix of luminance values Y as described is represented by Table 1. Extending the matrix of Y values to a larger size is straightforward. If the matrix of Y values is an eight-by-eight matrix, then the subsequent U and V matrices will each be four-by-four matrices.

D_{00}	D_{01}	D_{02}	. . . .	.	D_{07}
D_{10}	D_{11}	D_{12}	. . . .	.	D_{17}
.	.	.	. . . .	.	.
.	.	.	. . . .	.	.
.	.	.	. . . .	.	.
D_{70}	D_{71}	. . . .	. . . .	.	D_{77}

Table 1.

The order of the Y values supplied to the discrete wavelet transform circuit 122 is D_{00}, D_{01}, . . . D_{07} in the first row, then D_{10}, D_{11}, . . . D_{17} in the second row, and so forth row by row through the values in Table 1. Multiplexer 510 in Figure 5 is controlled by control block 506 to couple this stream of data values to the row convolver CONV_ROW block 502. The row convolver CONV_ROW block 502 performs a row convolution of the row data values D_{00}, D_{01}, D_{02}, ... D_{07} with a high pass four coefficient quasi-Daubechies digital filter G = (d, c, -b, a) and a low pass four coefficient quasi-Daubechies digital filter H = (a, b, c, -d) where a = 11/32, b = 19/32, c = 5/32, d = 3/32. The coefficients a, b, c, d are related to a four coefficient Daubechies wavelet as described in the copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression".

The operation of CONV_ROW block 502 on the data values of Table 1 is explained with reference to Figures 6, 8, 9, 10 and 11. Figure 8 is a detailed block diagram of the wavelet transform multiplier circuit 602 of the CONV_ROW block. Figure 9 is a detailed block diagram of the row wavelet transform circuit 604 of the CONV_ROW block. Figure 10 shows a sequence of control signals supplied by the control block 506 of Figure 5 to the row wavelet transform circuit 604 of Figure 9. This sequence of control signals effects a forward one dimensional wavelet transform on the rows of the matrix Table 1. The wavelet transform multiplier circuit 602 of Figure 8 comprises combinatorial logic which multiplies each successive input data value x by various scaled combinations of coefficients 32a, 32b, 32c, and 32d. This combinational-logic block comprises shift registers 802, 804, 806, and 808 which shift the multibit binary input data value x to the left by 1, 2, 3, and 4 bits, respectively. Various combinations of these shifted values, as well as the input value x itself, are supplied to multibit adders 810, 812, 814, 816, and 818. The data outputs 32dx, 32(c-d)x, 32cx, 32ax, 32(a+b)x, 32bx, and 32(c+d)x are therefore available to the row wavelet transform circuit 604 on separate sets of leads as shown in detail in Figures 6 and 9.

The row wavelet transform circuit 604 of Figure 9 comprises sets of multiplexers, adders, and delay elements. Multiplexer mux1 902, multiplexer mux2 904, and multiplexer mux3 906 pass selected ones of the data outputs of the wavelet transform multiplier circuit 602 of Figure 8 as determined by control signals on leads 546 from CONTROL block 506 of Figure 5. These control signals on leads 546 are designated muxsel(1), muxsel(2), and muxsel(3) on Figure 9. The remainder of the control signals on leads 546 supplied from CONTROL block 506 to the row wavelet transform circuit 604 comprise andsel(1), andsel(2), andsel(3), andsel(4), addsel(1), addsel(2), addsel(3), addsel(4), muxandsel(1), muxandsel(2), muxandsel(3), centermuxsel(1) and centermuxsel(2).

Figure 10 shows values of the control signals at different times during a row convolution of the forward transform. For example, at time t=0, the control input signal to multiplexer mux2 904, muxsel(2), is equal to 2. Multiplexer mux2 904 therefore couples its second input leads carrying the value 32(a+b)x to its output leads. Each of multiplexers 908, 910, 912, and 914 either passes the data value on its input leads, or passes a zero, depending on the value of its control signal. Control signals andsel(1) through andsel(4) are supplied to select input leads of multiplexers 908, 910, 912, and 914, respectively. Multiplexers 916, 918, and 920 have similar functionality. The outputs of multiplexers 916, 918, and 920 depend on the values of control signals muxandsel(1) through muxandsel(3), respectively. Multiplexers 922 and 924 pass either the value on their "left" input leads or the value on their "right" input leads, as determined by control select inputs centermuxsel(1) and centermuxsel(2), respectively. Adder/subtractors 926, 928, 930, and 932 either pass the sum or the difference of the values on their left and right input leads, depending on the values of the control signals addsel(1) through addsel(4), respectively. Elements 934, 936, 938, and 940 are one-cycle delay elements which output the data values that were at their respective input leads during the previous time period.

Figure 11 is a diagram of a data flow through the row convolver CONV_ROW 502 during a forward transform operation on the data values of Table 1 when the control signals 546 controlling the row convolver CONV_ROW 502 are as shown in Figure 10. At the left hand edge of the matrix of the data values of Table 1, start forward low pass and start forward high pass filters G_S and H_S are applied in accordance with equations 22 and 24 of copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression" as follows:

$$32H_{00} = 32\{(a + b)D_{00} + cD_{01} - dD_{02}\}$$
$$32G_{00} = 32\{(c + d)D_{00} - bD_{01} + aD_{02}\}$$

The row wavelet transform circuit of Figure 9 begins applying these start forward low and high pass filters when the control signals for this circuit assume the values at time t=0 as illustrated in Figure 10.

At time t=0, muxsel(2) has a value of 2. Multiplexer mux2 904 therefore outputs the value $32(a+b)D_{00}$ onto its output leads. Muxsel(3) has a value of 3 so multiplexer mux3 906 outputs the value $32(c+d)D_{00}$ into its output leads. Because the control signals andsel(2) and andsel(3) cause multiplexers 910 and 912 to output zeros at t=0 as shown in Figure 10, the output leads of adder/subtractor blocks 928 and 930 carry the values $32(a+b)D_{00}$ and $32(c+d)D_{00}$, respectively, as shown in Figure 11. These values are supplied to the input leads of delay elements 936 and 938. Delay elements 936 and 938 in the case of the row transform are one time unit delay elements. The control signals centermuxsel(1) and centermuxsel(2) have no effect at t=0, because control signals andsel(2) and andsel(3) cause multipliers 910 and 912 to output zeros.

At time t=1, input data value x is the data value D_{01}. Control signal muxsel(2) is set to 1 so that multiplexer mux2 904 outputs the value $32bD_{01}$. The select signal centermuxsel(1) for adder/subtractor block 922 is set to pass the value on its right input leads. The value $32(c+d)D_{00}$, the output of adder/subtractor block 930 at t=0, is therefore passed through multiplexer mux4 922 due to the one time unit delay of delay element 938. The control signal andsel(2) is set to pass, so the two values supplied to the adder/subtractor block 928 are $32(c+d)D_{00}$ and $32bD_{01}$. Because the control signal addsel(2) is set to subtract, the value output by adder/sub-

tractor block 928 is $32\{(c+d)D_{00}-bD_{01}\}$ as shown in Figure 11. Similarly, with the values of control signals centermuxsel(2), andsel(3), muxsel(3), muxandsel(2)-, and addsel(3) given in Figure 10, the value output by adder/subtractor block 930 is $32\{(a+b)D_{00} + cD_{01}\}$ as shown in Figure 11.

At time t=2, input data value x is data value D_{02}. The control signals andsel(1), muxsel(1), and muxandsel(1) are set so that the inputs to adder/subtractor block 926 are $32aD_{02}$ and $32\{(c+d)D_{00}-bD_{01}\}$. The value $32\{(c+d)D_{00}-bD_{01}\}$ was the previous output from adder/subtractor block 928. Because control signal addsel(1) is set to add as shown in Figure 10, the output of block 926 is $32\{(c+d)D_{00}-bD_{01}+aD_{02}\}$ as shown in Figure 11. Similarly, with the value of control signals addsel(4), andsel(4) and muxandsel(3), the value output by adder/subtractor block 932 is $32\{(a+b)D_{00} + cD_{01} - dD_{02}\}$ as shown in Figure 11.

As illustrated in Figure 10, output leads OUT2 (which are the output leads of delay element 940) carry a value of $32H_{00}$ at time t=3. The value $32\{(a+b)D_{00}-bD_{01}+aD_{02}\}$ is equal to $32H_{00}$ because $32H_{00}=32\{(a+b)D_{00}+cD_{01}-dD_{02}\}$ as set forth above. Similarly, output leads OUT1 (which are the output leads of delay element 934) carry a value of $32G_{00}$ at t=3 because output leads of block 926 have a value of $32\{(c+d)D_{00}-bD_{01}+aD_{02}\}$ one time period earlier. Because $32H_{00}$ precedes $32G_{00}$ in the data stream comprising the high and low pass components in a one-dimensional row convolution, delay element 606 is provided in the CONV_ROW row convolver of Figure 6 to delay $32G_{00}$ so that $32G_{00}$ follows $32H_{00}$ on the leads which are input to the multiplexer 608. Multiplexer 608 selects between the left and right inputs shown in Figure 6 as dictated by the value mux_608, which is provided on one of the control leads 546 from control block 506. The signal mux_608 is timed such that the value $32H_{00}$ precedes the value $32G_{00}$ on the output leads of multiplexer 608.

The output leads of multiplexer 608 are coupled to a variable shift register 610 as shown in Figure 6. The function of the variable shift register 610 is to normalize the data values output from the CONV_ROW block by shifting the value output by multiplexer 608 to the right by m_row bits. In this instance, for example, it is desirable to divide the value output of multiplexer 608 by 32 to produce the normalized values H_{00} and G_{00}. To accomplish this, the value m_row provided by control block 506 via one of the control leads 546 is set to 5. The general rule followed by the control block 506 of the discrete wavelet transform circuit is to: (1) set m_row equal to 5 to divide by 32 during the forward transform, (2) set m_row equal to 4 to divide by 16 during the middle of a row during an inverse transform, and (3) set m_row equal to 3 to divide by 8 when generating a start or end value of a row during the inverse transform. In the example being described, the start values of a transformed row during a forward transform are being generated, so m_row is appropriately set equal to 5.

As illustrated in Figure 10, the centermuxsel(1) and centermuxsel(2) control signals alternate such that the values on the right and the left input leads of multiplexers 922 and 924 are passed to their respective output leads for each successive data value convolved. This reverses data flow through the adder/subtractor blocks 928 and 930 in alternating time periods. In time period t=0, for example, Figure 11 indicates that the value $32aD_{01}$ in the column designated "Output of Block 926" in time period t=1 is added to $32bD_{02}$ to form the value $32\{aD_{01} + bD_{02}\}$ in the column designated "Output of Block 928" at time t=2. Then, in time period t=3, the value $32\{dD_{01} + cD_{02}\}$ in the column designated "Output of Block 930" is added to $32bD_{03}$ to form the value $32\{dD_{01} + cD_{02} - bD_{03}\}$ in the column designated "Output of Block 928".

Accordingly, in time period t=2, the two values supplied to block 928 are $32bD_{02}$ and the previous output from block 926, $32bD_{01}$. Because addsel(2) is set to add as shown in Figure 10, the value output by block 928 is $32(aD_{01} + bD_{02})$.

Similarly, the output of block 930 is $32(dD_{01} + cD_{02})$. In this way it can be seen the sequence of control signals in Figure 10 causes the circuit of Figure 9 to execute the data flow in Figure 11 to generate, after passage through multiplexer mux 608 and shift register 610 with m_row set equal to 5, the low and high pass non-boundary components H_{01}, G_{01}, H_{02}, and G_{02}. To implement the end forward low and high pass filters beginning at t=7 when the last data value of the first row of Table 1, D_{07}, is input to the row convolver, the control signal muxsel(2) is set to 3, so that $32(b-a)D_{07}$ is passed to block 928. Control signal muxsel(3) is set to 4, so that $32(c-d)D_{07}$ is passed to block 930. Control signal addsel(2) is set to subtract and control signal addsel(3) is set to add. Accordingly, the output of adder/subtractor 928 is $32(dD_{06}+cD_{06}-(b-a)D_{07})$. Similarly, the output of adder/subtractor 930 is $32(aD_{06} + bD_{06} + (c-d)D_{07})$.

As shown in Figure 11, these values are output from blocks 926 and 932 at the next time period when t=8 by setting muxandsel(1) and muxandsel(3) to be both zero so that adder/subtractor blocks 926 and 932 simply pass the values unchanged. Delay elements 934 and 940 cause the values $32G_{03}$ and $32H_{03}$ to be output from output leads OUT1 and OUT2 at time t=9. Multiplexer 608, as shown in Figure 6, selects between the output of delay unit 606 and the OUT2 output as dictated by CONTROL block 506 of Figure 5. Shift register 610 then normalizes the output as described previously, with m_row set equal to 5 for the end of the row. The resulting values G_{03} and H_{03} are the values output by the end low pass and end high pass forward transform digital filters in accordance with equations 26 and 28 of copending Patent Cooperation Treaty (PCT) application filed March

30, 1994, entitled "Data Compression and Decompression". Thus, a three coefficient start forward transform low pass filter and a three coefficient start forward transform high pass filter have generated the values H_{00} and G_{00}. A four coefficient quasi-Daubecheis low pass forward transform filter and a four coefficient quasi-Daubecheis high pass forward transform filter have generated the values H_{01} ... G_{02}. A three coefficient end forward transform low pass filter and a three coefficient end forward transform high pass filter have generated the values H_{03} and G_{03}.

The same sequence is repeated for each of the rows of the matrix in Table 1. In this way, for each two data values input there is one high pass (G) data value generated and there is one low pass (H) data value generated. The resulting output data values of CONV_ROW block 502 are shown in Figure 12.

As illustrated in Figure 5, the values output from row convolver CONV_ROW block 502 are passed to the column convolver CONV_COL block 504 in order to perform column convolution using the same filters in accordance with the method set forth in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression".

Figure 7 is a block diagram of the column convolver CONV_COL block 504 of Figure 5. The CONV_COL block 504 comprises a wavelet transform multiplier circuit 702, a column wavelet transform circuit 704, a multiplexer 708, and a variable shift register 710. In general, the overall operation of the circuit shown in Figure 7 is similar to the overall operation of the circuit shown in Figure 6. The wavelet transform multiplier circuit 702 of the column convolver is identical to the wavelet transform multiplier circuit 602 of Figure 6. The dashed line in Figure 8. Therefore, is designated with both reference numerals 602 and 702.

Figure 13 is a detailed block diagram of the column wavelet transform circuit 704 of Figure 7 of the column convolver. The CONV_COL block 504, as shown in Figure 13, is similar to the CONV_ROW block 502, except that the unitary delay elements 934, 936, 938, and 940 of the CONV_ROW block 502 are replaced by "line delay" blocks 1334, 1336, 1338, and 1340, respectively. The line delay blocks represent a time delay of one row which, in the case of the matrix of the presently described example, is eight time units. In some embodiments in accordance with the present invention, the line delays are realized using random access memory (RAM).

To perform a column convolution on the values of the matrix of Figure 12, the first three values H_{00}, H_{10}, H_{20} of the first column are processed to generate, after a bit shift in shift register 710 of Figure 7, low and high pass values HH_{00} and HG_{00} of Figure 16. The first three values G_{00}, G_{10}, G_{20} of the second column of the matrix of Figure 12 are then processed to likewise produce GH_{00} and GG_{00}, and so on, to produce the top two rows of values of the matrix of Figure 16. Three values in each column are processed because the start low and high pass filters are three coefficient filters rather than four coefficient filters.

Figure 14 is a diagram illustrating control signals which control the column convolver during the forward transform of the data values of Figure 12. Figure 15 is a diagram illustrating data flow through the column convolver. Corresponding pairs of data values are output from line delays 1334 and 1340 of the column wavelet transform circuit 704. For this reason, the low pass filter output values are supplied from the output leads of the adder/subtractor block 1332 at the input leads of line delay 1340 rather than from the output leads of the line delay 1340 so that a single transformed data value is output from the column wavelet transform circuit in each time period. In Figure 14, output data values $32HH_{00}$ ... $32GH_{03}$ are output during time periods t=16 to t=23 whereas output data values $32HG_{00}$ ... $32GG_{03}$ are output during time periods t=24 to t=31, one line delay later. After being passed through multiplexer 708 and variable shift register 710 of Figure 7, the column convolved data values HH_{00} ... GH_{03} and HG_{00} ... GG_{03} are written to memory unit 116 under the control of the address generator. After all the data values of Figure 16 are written to memory unit 116, an octave 0 sub-band decomposition exists in memory unit 116.

To perform the next octave of decomposition, only the low pass component HH values in memory unit 116 are processed. The HH values are read from memory unit 116 and passed through the CONV_ROW block 502 and CONV_COL block 504 as before, except that the control signals for control block 506 are modified to reflect the smaller matrix of data values being processed. The line delay in the CONV_COL block 504 is also shortened to four time units because there are now only four low pass component HH values per row. The control signals to accomplish the octave 1 forward row transform on the data values in Figure 16 are shown in Figure 17. The corresponding data flow for the octave 1 forward row transform is shown in Figure 18. Likewise, the control signals to accomplish the octave 1 forward column transform are shown in Figure 19, and the corresponding data flow for the octave 1 forward column transform is shown in Figure 20.

The resulting HHHH, HHHG, HHGH, and HHGG data values output from the column convolver CONV_COL block 504 are sent to memory unit 116 to overwrite only the locations in memory unit 116 storing corresponding HH data values as explained in connection with Figures 17 and 18 of copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression". The result is an octave 1 sub-band decomposition stored in memory unit 116. This process can be performed on

large matrices of data values to generate sub-band decompositions having as many octaves as required. For ease of explanation and illustration, control inputs and dataflow diagrams are not shown for the presently described example for octaves higher than octave 1. However, control inputs and dataflows for octaves 2 and above can be constructed given the method described in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression" along with the octave 0 and octave 1 implementation of that method described above.

Figure 21 illustrates a block diagram of one possible embodiment of control block 506 of Figure 5. Control block 506 comprises a counter 2102 and a combinatorial logic block 2104. The control signals for the forward and inverse discrete wavelet transform operations, as shown in Figures 10, 14, 17, 19, 22, 24, 26, and 28, are output onto the output leads of the combinatorial logic block 2104. The input signals to the control block 506 comprise the sync leads 201 which are coupled to A/D video decoder 110, the direction lead 538 which is coupled to REGISTERS block 536, and the image size leads 540 and 542 which are also coupled to REGISTERS block 536. The values of the signals on the register leads 538, 540, and 542 are downloaded to REGISTERS block 536 of the video encoder/decoder chip 112 from data bus 106 via register download bus 128. The output leads of control block 506 comprise CONV_ROW control leads 546, CONV_COL control leads 548, DWT control leads 550, 552, and 554, memory control leads 2106 and 2108, DWT address generator muxcontrol lead 556, DWT address generator read control leads 534, and DWT address generator write control leads 544.

Counter block 2102 generates the signals row_count, row_carry, col_count, col_carry, octave, and channel, and provides these signals to combinatorial logic block 2104. Among other operations, counter 2102 generates the signals row_count and row_carry by counting the sequence of data values from 0 up to ximage, where ximage represents the horizontal dimension of the image received on leads 540. Similarly, counter 2102 generates the signals col_count and col_carry by counting the sequence of data values from 0 up to yimage, where yimage represents the vertical dimension of the image received on leads 542. The inputs to combinatorial logic block 2104 comprise the outputs of counter block 2102 as well as the inputs direction, ximage, yimage and sync to control block 506. The output control sequences of combinatorial logic block 2104 are combinatorially generated from the signals supplied to logic block 2104.

After the Y data values of an image have been transformed, the chrominance components U and V of the image are transformed. In the presently described specific embodiment of the present invention, a 4:1:1 format of Y:U:V values is used. Each of the U and V matrices of data values comprises half the number of rows and columns as does the Y matrix of data values. The wavelet transform of each of these components of chrominance is similar to the transformation of the Y data values except the line delays in the CONV_COL are shorter to accommodate the shorter row length and the size of the matrices corresponding to the matrix of Table 1 is smaller.

Not only does the discrete wavelet transform circuit of Figure 5 transform image data values into a multi-octave sub-band decomposition using a forward discrete wavelet transformation, but the discrete wavelet transform circuit of Figure 5 can be used to perform a discrete inverse wavelet transform on transformed-image data to convert a sub-band decomposition back into the image domain. In one octave of an inverse discrete wavelet transform, the inverse column convolver 504 of Figure 5 operates on transformed-image data values read from memory unit 116 via leads 526, lines 528 and multiplexer mux2 512 and the inverse row convolver 502 operates on the data values output by the column convolver supplied via leads 524, lines 525 and multiplexer mux1 510.

Figures 22 and 23 show control signals and data flow for the column convolver 504 of Figure 5 when column convolver 504 performs an inverse octave 1 discrete wavelet transform on transformed-image data located in memory unit 116. As illustrated in Figure 23, the data value output from adder/subtractor block 1326 of Figure 13 at time t=4 is $32\{(b-a)HHHH_{00} + (c-d)HHHG_{00}\}$. The column convolver therefore processes the first two values $HHHH_{00}$ and $HHHG_{00}$ in accordance with the two coefficient start reconstruction filter (inverse transform filter) set forth in equation 52 of copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression". Subsequently, blocks 1332 and 1326 output values indicating that the column convolver performs the four coefficient odd and even reconstruction filters (interleaved inverse transform filters) of equations 20 and 19 of copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression". Fig. 23 illustrates that the column convolver performs the two coefficient end reconstruction filter (inverse transform filter) on the last two data values $HHHH_{10}$ and $HHHG_{10}$ (see time t=20) of the first column of transformed data values in accordance with equation 59 of copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression". The data values output from the column convolver of Figure 13 are supplied to the row convolver 502 of Figure 5 via lines 525 and multiplexer mux1 510.

Figures 24 and 25 show control signals and data flow for the row convolver 502 of Figure 5 when the row convolver performs an inverse octave 1 discrete wavelet transform on the data values output from the column

convolver. The column convolver 504 has received transformed values $HHHH_{00}$ ... $HHGH_{01}$ and so forth as illustrated in Fig. 23 and generated the values HHH_{00} ... HHG_{01} and so forth, as illustrated in Fig. 22, onto output leads 524. Row convolver 502 receives the values HHH_{00} ... HHG_{01} and so forth as illustrated in Fig. 24 and generates the values HH_{00}, HH_{01}, HH_{02} and so forth as illustrated in Fig. 24 onto output leads 520 of row convolver 502. The data flow of Fig. 25 indicates that the row convolver performs the start reconstruction filter on the first two data values of a row, performs the odd and even reconstruction filters on subsequent non-boundary data values, and performs the end reconstruction filter on the last two data values of a row. The HH data values output from row convolver 502 are written to memory unit 116 into the memory locations corresponding with the HH data values shown in Fig. 16.

To inverse transform the octave 0 data values in memory unit 116 into the image domain, the column convolver 504 and the row convolver 502 perform an inverse octave 0 discrete wavelet transform. Figures 26 and 27 show the control signals and the data flow for the column convolver 504 of Figure 5 when the column convolver performs an inverse octave 0 discrete wavelet transform on transformed image data values in memory unit 116. The data values output from the column convolver are then supplied to the row convolver 502 of Figure 5 via lines 528 and multiplexer mux1 510.

Figures 28 and 29 show control signals and data flow for the row convolver 502 of Figure 5 when the row convolver performs an inverse octave 0 discrete wavelet transform on the data output from the column convolver to inverse transform the transformed-image data back to the image domain. Column convolver 504 receives transformed values HH_{00} ... GH_{03} and so forth as illustrated in Fig. 27 and generates the values H_{00} ... G_{03} and so forth as illustrated in Fig. 26 onto output leads 524. Row convolver 502 receives the values H_{00} ... G_{03} and so forth, as illustrated in Fig. 28, and generates the inverse transformed data values D_{00}, D_{01}, D_{02} ... D_{07} and so forth, as illustrated in Fig. 28, onto output leads 520 of row convolver 502. The inverse transformed data values output from row convolver 502 are written to memory unit 114.

The control signals and the data flows of Figures 22, 23, 24, 25, 26, 27, 28 and 29 comprise the inverse transformation from octave 1 to octave 0 and from octave 0 back into image domain inverse transformed data values which are substantially the same as the original data values of the matrix Table 1. The control signals which control the row convolver and column convolver to perform the inverse transform are generated by control block 506. The addresses and control signals used to read data values from and write data values to memory units 116 and 114 are generated by the DWT address generator block 508 under the control of control block 506.

After the inverse wavelet transform of the Y matrix of transformed data values is completed, the U and V matrices of transformed data values are inverse transformed one after the other in a similar way to the way the Y matrix was inverse transformed.

Figure 30 is a block diagram of the DWT address generator block 508 of Figure 5. The DWT address generator block 508 supplies read and/or write addresses to the memory units 116 and 114 for each octave of the forward and inverse transform. The DWT address generator block 508 comprises a read address generator portion and a write address generator portion. The read address generator portion comprises multiplexer 3006, adder 3010, multiplexer 3002, and resettable delay element 3014. The write address generator portion likewise comprises multiplexer 3008, adder 3012, multiplexer 3004, and resettable delay element 3016. The DWT address generator is coupled to the control block 506 via control leads 534, 556, and 544, to memory unit 116 via address leads 3022, and to memory unit 114 via address leads 3020. The input leads of DWT address generator 508 comprise the DWT address generator read control leads 534, the DWT address generator write control leads 544, and the muxcontrol lead 534. The DWT address generator read control leads 534, in turn, comprise 6 leads which carry the values col_end_R, channel_start_R, reset_R, oct_add_factor_R, incr_R, base_u_R, and base_v_R. The DWT address generator write control leads 544, in turn, comprise leads which carry the values col_end_W, channel_start_W, reset_W, oct_add_factor_W, incr_W, base_u_W, and base_v_W. All signals contained on these leads are provided by control block 506. The output leads of DWT address generator block 508 comprise address leads 3022 which provide address information to memory unit 116, and address leads 3020 which provide address information to memory unit 114. The addresses provided on leads 3022 can be either read or write addresses, depending on the cycle of the DWT transform circuit 122 as dictated by control signal muxcontrol provided by control block 506 on lead 556. The addresses provided on leads 3020 are write-only addresses, because memory unit 114 is only written to by the DWT transform circuit 122.

Memory locations of a two-dimensional matrix of data values such as the matrices of Table 1, Figure 12 and Figure 16 may have memory location addresses designated 0, 1, 2 and so forth, the addresses increasing by one left to right across each row and increasing by one to skip from the right most memory location at the end of a row to the left most memory location of the next lower row. To address successive data values in a matrix of octave 0 data values, the address is incremented by one to read each new data value D from the

matrix.

For octave 1, addresses are incremented by two because the HH values are two columns apart as illustrated in Figure 16. The row number, however, is incremented by two rather than one because the HH values are located on every other row. The DWT address generator 508 in octave 1 therefore increments by two until the end of a row is reached. The DWT address generator then increments once by ximage + 2 as can be seen from Figure 16. For example, the last HH value in row 0 of Figure 16 is HH_{03} at memory address 6 assuming HH_{00} has an address of 0 and that addresses increment by one from left to right, row by row, through the data values of the matrix. The next HH value is in row two, HH_{10}, at memory address 16. The increment factor in a row is therefore incr = 2^{octave}. The increment factor at the end of a row is oct_add_factor = $(2^{octave} - 1) * ximage + 2^{octave}$ for octave ≥ 0, where ximage is the x dimension of the image.

In some embodiments, the transformed Y data values are stored in memory unit 116 from addresses 0 through (ximage * yimage - 1), where yimage is the y dimension of the matrix of the Y data values. The transformed U data values are then stored in memory unit 116 from address base_u up to base_v - 1, where:

$$base\_u = ximage * yimage$$

$$base\_v = ximage * yimage + \frac{ximage * yimage}{4}$$

Similarly, the transformed V data values are stored in memory unit 116 at addresses beginning at address base_v.

The operation of the read address generator portion in Figure 30 is representative of both the read and write portions. In operation, multiplexer base_mux 3002 of Figure 30 sets the read base addresses to be 0 for the Y channel, base_u_R for the U channel, and base_v_R for the V channel. Multiplexer 3002 is controlled by the control signals channel_start_R which signifies when each Y, U, V channel starts. Multiplexer mux 3006 sets the increment factor to be incr_R, or, at the end of each row, to oct_add_factor_R. The opposite increment factor is supplied to adder 3010 which adds the increment factor to the current address present on the output leads of delay elements 3014 so as to generate the next read address, next_addr_R. The next read address next_addr_R is then stored in the delay element 3014.

In some embodiments In accordance with the present invention, tables of incr_R and oct_add_factor_R for each octave are downloaded to REGISTERS block 536 on the video encoder/decoder chip 112 at initialization via download registers bus 128. These tables are passed to the control block 506 at initialization. To clarify the illustration, the leads which connect REGISTERS block 536 to control block 506 are not included in Figure 5. In other embodiments, values of incr_R and oct_add_factor_R are precalculated in hardware from the value of ximage using a small number of gates located on-chip. Because the U and V matrices have half the number of columns as the Y matrix, the U and V jump tables are computed with ximage replaced by $\frac{ximage}{2}$, a one bit shift. Because the tree encoder/decoder restricts ximage to be a multiple of $2^{(OCT+1)} > 2^{octave}$, the addition of 2^{octave} in the oct_add_factor is, in fact, concatenation. Accordingly, only the factor $(2^{octave-1}) * ximage$ must be calculated and downloaded. The jump tables for the U and V addresses can be obtained from the Y addresses by shifting this factor one bit to the right and then concatenating with 2^{octave}. Accordingly, appropriate data values of a matrix can be read from a memory storing the matrix and processed data values can be written back into the matrix in the memory to the appropriate memory locations.

Figures 31 and 32 are block diagrams of one embodiment of the tree processor/encoder-decoder circuit 124 of Figure 1. Figure 31 illustrates the circuit in encoder mode and Figure 32 illustrates the circuit in decoder mode. Tree processor/encoder-decoder circuit 124 comprises the following blocks: DECIDE block 3112, TP_ADDR_GEN block 3114, quantizer block 3116, MODE_CONTROL block 3118, Huffman encoder-decoder block 3120, buffer block 3122, CONTROL_COUNTER block 3124, delay element 3126, delay element 3128, and VALUE_REGISTERS block 3130.

The tree processor/encoder-decoder circuit 124 is coupled to FIFO buffer 120 via input/output data leads 130. The tree processor/encoder-decoder circuit 124 is coupled to memory unit 116 via an old frame data bus 3102, a new frame data bus 3104, an address bus 3106, and memory control buses 3108 and 3110. The VALUE_REGISTERS block 3130 of the tree processor/encoder-decoder circuit 124 is coupled to data bus 106 via a register download bus 128. Figures 31 and 32 illustrate the same physical hardware; the encoder and decoder configurations of the hardware are shown separately for clarity. Although two data buses 3104 and 3102 are illustrated separately in Figure 31 to facilitate understanding, the new and old frame data buses may actually share the same pins on video encoder/decoder chip 112 so that the new and old frame data are time multiplexed on the same leads 526 of memory unit 116 as illustrated in Figure 5. Control buses 3108 and 3110 of Figure 31 correspond with the control lines 2108 in Figure 5. The DWT address generator block 508 of the discrete wavelet transform circuit 122 and the tree processor address generator block 3114 of the tree processor/encoder-decoder circuit 124 access memory unit 116 therefore may use the same physical address, data and

control lines.

Figure 33 illustretes an embodiment of DECIDE block 3112. A function of DECIDE block 3112 is to receive a two-by-two block of data values from memory unit 116 for each of the old and new frames and from these two-by-two blocks of data values and from the signals on leads 3316, 3318, 3320 and 3322, to generate seven flags present on leads 3302, 3304, 3306, 3308, 3310, 3312 and 3314. The MODE_CONTROL block 3118 uses these flags as well as values from VALUE_REGISTERS block 3130 supplied via leads 3316, 3318 and 3320 to determine the mode in which the new two-by-two block will be encoded. The addresses in memory unit 116 at which the data values of the new and old two-by-two blocks are located and determined by the address generator TP_ADDR_GEN block 3114.

The input signal on register lead 3316 is the limit value output from VALUE_REGISTERS block 3130. The input signal on register leads 3318 is the qstep value output from VALUE_REGISTERS block 3130. The input signal on register lead 3320 is the compare value output from VALUE_REGISTERS block 3130. The input signal on register lead 3322 is the octave value generated by TP_ADDR_GEN block 3114 as a function of the current location in the tree of the sub-band decomposition. As described in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression" at equations 62-71, the values of the flags new_z, nz_flag, origin, noflag, no_z, oz_flag, and motion, produced on leads 3302, 3304, 3306, 3308, 3310, 3312, and 3314, respectively, are determined in accordance with the following equations:

$$nz = \sum_{0 \le x, y \le 1} |new[x][y]| \qquad \text{(equ. 1)}$$

$$oz = \sum_{0 \le x, y \le 1} |old[x][y]| \qquad \text{(equ. 2)}$$

$$no = \sum_{0 \le x, y \le 1} |new[x][y] - old[x][y]| \qquad \text{(equ. 3)}$$

$$nz\_flag = nz < limit \quad \text{(equ. 4)}$$

$$noflag = no < compare \quad \text{(equ. 5)}$$

$$origin = nz \le no \quad \text{(equ. 6)}$$

$$motion = ((nz + oz) << octave) \le no \quad \text{(equ. 7)}$$

$$new\_z = |new[x][y]| < qstep, \quad for\ 0 \le x, y \le 1 \quad \text{(equ. 8)}$$

$$no\_z = |new[x][y] - old[x][y]| < qstep, \quad for\ 0 \le x, y \le 1 \quad \text{(equ. 9)}$$

$$oz\_flag = old[x][y] = 0, \quad for\ all\ 0 \le x, y, \le 1 \quad \text{(equ. 10)}$$

The DECIDE block 3112 comprises subtractor block 3324, absolute value (ABS) blocks 3326, 3328, and 3330, summation blocks 3332, 3334, and 3336, comparator blocks 3338, 3340, 3342, 3344, 3346, 3350, and 3352, adder block 3354, and shift register block 3356. The value output by ABS block 3326 is the absolute value of the data value new[x][y] on leads 3104. Similarly, the value output by ABS block 3328 is the absolute value of the data value old[x][y] on leads 3102. The value output by ABS block 3330 is the absolute value of the difference between the data values new[x][y] and old[x][y]. Comparator 3338, coupled to the output leads of ABS block 3326, unasserts new_z flag on lead output 3302 if qstep is less than the value output by block 3326. Block 3332 sums the last four values output from block 3326 and the value output by block 3332 is supplied to comparator block 3340. Comparator block 3340 compares this value to the value of limit 3316. The flag nz_flag 3304 is asserted on lead 3304 if limit is greater than or equal to the value output by block 3332. This value corresponds to nz_flag in equation 4. Summation block 3334 similarly sums the four most recent values output by block 3328. The values outputs by blocks 3332 and 3334 are added together by block 3354, the values output by block 3354 being supplied to shift register block 3356. The shift register block 3356 shifts the value received to the left by octave bits. Summation block 3336 adds the four most recent values output by block 3330. Comparator block 3342 compares the value output by block 3332 to the value output by block

333€ and asserts the motion flag in accordance with equation 7. The origin flag on output lead 3306 is asserted when the value output by block 3332 is less than the value output by 3336. This value corresponds to origin in equation 6 above. The value output by block 3336 is compared to the value compare by block 3344 such that flag noflag is asserted when compare is greater than the value output from block 3336. Block 3346 compares the value output by block 3330 to the value qstep such that flag no_z is unasserted when qstep is less. This corresponds to flag no_z in equation 9. The old input value on leads 3102 is compared to the value 0 by block 3350 such that flag oz_flag on lead 3312 is asserted when each of the values of the old block is equal to 0. This corresponds to oz_flag in equation 10 above. The seven flags produced by the DECIDE block of Figure 33 are passed to the MODE_CONTROL block 3118 to determine the next mode.

The tree processor/encoder-decoder circuit 124 of Figure 31 comprises delay elements 3126 and 3128. Delay element 3126 is coupled to the NEW portion of memory unit 116 via new frame data bus 3104 to receive the value new[x][y]. Delay element 3128 is coupled to the OLD portion of memory unit 116 via old frame data bus 3102 to receive the value old[x][y]. These delay elements, which in some embodiments of the invention are implemented in static random access memory (SRAM), serve to delay their respective input values read from memory unit 116 for four cycles before the values are supplied to quantizer block 3116. This delay is needed because the DECIDE block 3112 introduces a four-cycle delay in the dataflow as a result needing to read the four most recent data values before the new mode in which those data values will be encoded is determined. The delay elements therefore synchronize signals supplied to quantizer block 3116 by the MODE_CONTROL block 3118 with the values read from memory unit 116 which are supplied to quantizer block 3116.

The tree processor/encoder-decoder circuit 124 of Figures 31 and 32 comprises a VALUE_REGISTERS block 3130. The VALUE_REGISTERS block 3130 serves the function of receiving values from an external source and asserting these values onto leads 3316, 3318, 3320, 3132, 3134 and 3136, which are coupled to other blocks in the tree processor/encoder-decoder 124. In the presently described embodiment the external source is data bus 106 and VALUE_REGISTERS block 3130 is coupled to data bus 106 via a download register bus 128. Register leads 3316 carry a signal corresponding to the value of limit and are coupled to DECIDE block 3112 and to MODE_CONTROL block 3118. Register leads 3318 carry signals indicating the value of qstep and are coupled to DECIDE block 3112 and to MODE_CONTROL block 3118. Register leads 3320 carry signals indicating the value of compare and are coupled to DECIDE block 3112 and to MODE_CONTROL block 3118. Register leads 3132 carry signals indicating the value of ximage and are coupled to TP_ADDR_GEN block 3114 and to MODE_CONTROL block 3118. Register leads 3134 carry signals indicating the value of yimage and are coupled to TP_ADDR_GEN block 3114 and to MODE_CONTROL block 3118. Register lead 3136 carries a signal corresponding to the value of direction and is coupled to TP_ADDR_GEN block 3114, MODE_CONTROL block 3118, buffer block 3122, Huffman encoder-decoder block 3120, and quantizer block 3116. To clarify the illustration, only selected ones of the connections between the VALUE_REGISTERS block 3130 and other blocks of the tree processor/encoder-decoder circuit 124 are illustrated in Figures 31 and 32. VALUE_REGISTERS block 3130 is, in some embodiments, a memory mapped register addressable from bus 106.

Figure 34 is a block diagram of an embodiment of address generator TP_ADDR_GEN block 3114 of Figure 32. The TP_ADDR_GEN block 3114 of Figure 34 generates addresses to access selected two-by-two blocks of data values in a tree of a sub-band decomposition using a counter circuit (see Figures 27-29 of copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression" and the corresponding text). Figure 34 illustrates a three-octave counter circuit. The signals supplied to TP_ADDR_GEN block 3114 are provided by MODE_CONTROL block 3118, CONTROL_COUNTER block 3124, and VALUE_REGISTERS block 3130. MODE_CONTROL block 3118 is coupled to TP_ADDR_GEN block 3114 by leads 3402 which carry the three bit value new_mode. CONTROL_COUNTER 3124 is coupled to TP_ADDR_GEN block 3114 by leads 3404 and 3406 which carry signals read_enable and write_enable, respectively. VALUE_REGISTER block 3130 is coupled to TP_ADDR_GEN block 3114 by register leads 3132 which carry a signal indicating the value of ximage. The output leads of TP_ADDR_GEN block 3114 comprise tree processor address bus 3106 and octave leads 3322. The address generator TP_ADDR_GEN block 3114 comprises a series of separate counters: counter TreeRoot_x 3410, counter TreeRoot_y 3408, counter C3 3412, counter C2 3414, counter C1 3416, and counter sub_count 3418. TP_ADDR_GEN block 3114 also comprises CONTROL_ENABLE block 3420, multiplexer 3428, multiplexer 3430, NOR gate 3436, AND gates 3422, 3424 and 3426, AND gates 3428, 3430 and 3432, multiplier block 3432 and adder block 3434.

Counter TreeRoot_x 3410 counts from 0 up to $\frac{ximage}{2^{OCT+1}} - 1$ and counter TreeRoot_y 3408 counts from 0 up to $\frac{yimage}{2^{OCT+1}} - 1$, where OCT is the maximum number of octaves in the decomposition. Counters C3, C2, C1, and sub_count are each 2-bit counters which count from 0 up to 3, and then return to 0. Each of these counters

takes on its next value in response to a respective count enable control signal supplied by CONTROL_ENABLE block 3420. Figure 34 shows count enable control signals x_en, y_en, c3_en, c2_en, c1_en, and sub_en, being supplied to the counters TreeRoot_x, TreeRoot_y, C3, C2, C1 and sub_count, respectively. When one of the counters reaches its maximum value, the counter asserts a carry out signal back to the CONTROL_ENABLE block 3420. These carry out signals are denoted in Figure 34 as x_carry, y_carry, c3_carry, c2_carry, c1_carry, and sub_carry.

CONTROL_ENABLE block 3420 responds to input signal new_mode on leads 3402 and to the carry out signals to generate the counter enable signals. The octave signal output by CONTROL_ENABLE is the value of the octave of the transform of the data values currently being addressed. The c1_carry, c2_carry, and c3_carry signals are logically ANDed with the write_enable signal supplied from CONTROL_COUNTER block 450 before entering the CONTROL_ENABLE block 3420. This AND operation is performed by AND gates 3422, 3424, and 3426 as shown in Figure 34. The counter enable signals from CONTROL_ENABLE block 3420 are logically ANDed with the signal resulting from the logical ORing of read_enable and write_enable by OR gate 3436. These ANDing operations are performed by AND gates 3428, 3430, and 3432 as shown in Figure 34. AND gates 3422, 3424, 3426, 3428, 3430, and 3432 function to gate the enable and carry signals with the read_enable and write_enable signals such that the address space is cycled through twice per state, once for reading and once for writing.

The CONTROL_ENABLE block 3420 outputs the enable signals enabling selected counters to increment when the count value reaches 3 in the case of the 2-bit counters 3412, 3414, and 3418, or when the count value reaches $\frac{ximage}{2^{OCT+1}} - 1$ in the case of TreeRoot_x 3410, or when the count value reaches $\frac{yimage}{2^{OCT+1}} - 1$ in the case of TreeRoot_y 3408. The resulting x and y addresses of a two-by-two block of data values of a given octave in a matrix of data values are obtained from the signals output by the various counters as follows:

For octave = 0:

x = TreeRoot_x C3(2) C2(2) C1(2) sub_count(2) (equ. 11)

y = TreeRoot_y C3(1) C2(1) C1(1) sub_count(1) (equ. 12)

For octave = 1:

x = TreeRoot_x C3(2) C2(2) sub_count(2) 0 (equ. 13)

y = TreeRoot_y C3(1) C2(1) sub_count(1) 0 (equ. 14)

For octave = 2:

x = TreeRoot_x C3(2) sub_count(2) 0 0 (equ. 15)

y = TreeRoot_y C3(1) sub_count(1) 0 0 (equ. 16)

Figure 34 and equations 11-16 illustrate how the x and y address component values are generated by multiplexers 3428 and 3430, respectively, depending on the value of octave. The (2) in equations 11-16 denotes the least significant bit of a 2-bit counter whereas the (1) denotes the most significant bit of a 2-bit counter. TreeRoot_x and TreeRoot_y are the multibit values output by counters 3410 and 3408, respectively. The output of multiplexer 3430 is supplied to multiplier 3432 so that the value output by multiplexer 3430 is multiplied by the value ximage. The value output by multiplier 3432 is added to the value output by multiplexer 3428 by adder block 3434 resulting in the actual address being output onto address bus 3106 and to memory unit 116.

Appendix A discloses one possible embodiment of CONTROL_ENABLE block 3420 of a three octave address generator described in the hardware description language VHDL. An overview of the specific implementation given in this VHDL code is provided below. The CONTROL_ENABLE block 3420 illustrated in Figure 34 and disclosed in Appendix A is a state machine which allows trees of a sub-band decomposition to be ascended or descended as required by the encoding or decoding method. The CONTROL_ENABLE block 3420 generates enable signals such that the counters generate four addresses of a two-by-two block of data values at a location in a tree designated by MODE_CONTROL block 3118. Instructions from the MODE_CONTROL block 3118 are read via leads 3402 which carry the value new_mode. Each state is visited for four consecutive cycles so that the four addresses of the block are output by enabling the appropriate counter C3 3412, C2 3414 or C1 3416. Once the appropriate counter reaches a count of 3, a carry out signal is sent back to CONTROL_ENABLE block 3420 so that the next state is entered on the next cycle.

Figure 35 is a state table for the TP_ADDR_GEN block 3114 of Figure 34 when the TP_ADDR_GEN block 3114 traverses all the blocks of the tree illustrated in Figure 36. Figure 35 has rows, each of which represents the generation of four address values of a block of data values. The (0-3) designation in Figure 35 represents the four values output by a counter. The names of the states (i.e, up0, up1, down1) do not indicate movement up or down the blocks of a tree but rather correspond with state names present in the VHDL code of Appendix A. (In Appendix A, the states down1, down2 and down3 are all referred to as down1 to optimize the implementation.) The state up0 in the top row of Figure 35, for example, corresponds to addressing the values of two-by-two block located at the root of the tree of Figure 36. In the tree of Figure 36 there are three octaves. After

these four addresses of the two-by-two block at the root of the tree are generated, the tree may be ascended to octave 1 by entering the state up1.

Figure 36 illustrates a complete traversal of all the data values of one tree of a 3-octave sub-band decomposition as well as the corresponding states of the CONTROL_ENABLE block of Figure 35. One such tree exists for each of the "GH", "HG" and "GG" sub-bands of a sub-band decomposition.

First, before a tree of the sub-band decomposition is traversed, all low pass HHHHHH component values of the decomposition are addressed by setting counter sub_count to output 00. Counter C3 3412 is incremented through its four values. Counter TreeRoot_x is then incremented and counter C3 3412 is incremented through its four values again. This process is repeated until TreeRoot_x reaches its maximum value. The process is then repeated with TreeRoot_y being incremented. In this manner, all HHHHHH low pass components are accessed. Equations 15 and 16 are used to compute the addresses of the HHHHHH low pass component data values.

Next, the blocks of the "GH" subband of a tree given by TreeRoot_x and TreeRoot_y are addressed. This "GH" subband corresponds to the value sub_count = 10 (sub_count (1) = 1 and sub_count (2) = 0). The up0 state shown in Figure 35 is used the generate the four addresses of the root block of the "GH" tree in accordance with equation 15. The up1 state shown in Figure 35 is then used such that addresses corresponding to equations 13 and 14 are computed to access the desired two-by-two block of data values in octave 1. The four two-by-two blocks in octave 0 are then accessed in accordance with equations 11 and 12. With TreeRoot_x and TreeRoot_y and sub_count untouched, the states zz0, zz1, zz2 and zz3 are successively entered, four addresses being generated in each state. After each one of these four states is exited, the C2 counter 3414 is incremented by CONTROL_ENABLE block 3420 via the c2_en signal once in order to move to the next octave 0 block in that branch of the tree. After incrementing in state zz3 is completed, the left hand branch of the tree is exhausted. To move to the next two-by-two block, the C3 counter 3412 is incremented and the C2 counter 3414 is cycled through its four values to generate the four addresses of the next octave 1 block in state down1 in accordance with equations 13 and 14. In this way, the TP_ADDR_GEN block 3114 generates the appropriate addresses to traverse the tree in accordance with instructions received from MODE_CONTROL block 3118. When the traversal of the "GH" sub-band tree is completed, the traversal of the sub-band decomposition moves to the corresponding tree of the next sub-band without changing the value of TreeRoot_x and TreeRoot_y. Accordingly, a "GH" "HG" and "GG" family of trees are traversed. After all the blocks of the three sub-band trees have been traversed, the TreeRoot_x and TreeRoot_y values are changed to move to another family of sub-band trees.

To move to the next family of sub-band trees, the counter TreeRoot_x 3410 is incremented, and the C3 3412, C2 3414, C1 3416 counters are returned to 0. The process of traversing the new "GH" tree under the control of the MODE_CONTROL block 3118 proceeds as before. Similarly, the corresponding "HG" and "GG" trees are traversed. After TreeRoot_x 3410 reaches its final value, a whole row of tree families has been searched. The counter TreeRoot_y 3408 is therefore incremented to move to the next row of tree families. This process may be continued until all of the trees in the decomposition have been processed.

The low pass component HHHHHH (when sub_count = 00) does not have a tree decomposition. In accordance with the present embodiment of the present invention, all of the low pass component data values are read first as described above and are encoded before the tree encoder reads and encodes the three subbands. The address of the data values in the HHHHHH subband are obtained from the octave 3 x and y addresses with sub_count = 00. Counters C3 3412, TreeRoot_x 3410, and TreeRoot_y 3408 run through their respective values. After the low pass component data values and all of the trees of all the sub-bands for the Y data values have been encoded, the tree traversal method repeats on the U and V data values.

Although all the blocks of the tree of Figure 36 are traversed in the above example of a tree traversal, the MODE_CONTROL block 3118 may under certain conditions decide to cease processing data values of a particular branch and to move to the next branch of the tree as set forth in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression". This occurs, for example, when the value new_mode output by the MODE_CONTROL block 3118 indicates the mode STOP. In this case, the state machine of CONTROL_ENABLE block 3420 will move to, depending on the current location in the tree, either the next branch, or, if the branch just completed is the last branch of the last tree, the next tree.

Figure 34 illustrates control signal inputs read_enable and write_enable being supplied to TP_ADDR_GEN block 3114. These enable signals are provided because the reading of the new/old blocks and the writing of the updated values to the old frame memory occur at different times. To avoid needing two address generators, the enable signals of the counters C3 3412, C2 3414, and C1 3416 are logically ANDed with the logical OR of the read_enable and write_enable signals. Similarly, the carryout signals of these counters are logically ANDed with the write_enable signal. During time periods when the new/old blocks are read from memory, the

read_enable signal is set high and the write_enable signal is set low. This has the effect of generating the addresses of a two-by-two block, but disabling the change of state at the end of the block count. The counters therefore return to their original values they had the start of the block count so that the same sequence of four address values will be generated when the write_enable signal is set high. This time, however, the carry out is enabled into the CONTROL_ENABLE block 3420. The next state is therefore entered at the conclusion of the block count. In this manner, the address space is cycled through twice per state, once for reading and once for writing.

Figure 37 is a block diagram of one embodiment of quantizer block 3116 of Figure 31. As shown in Figure 31, quantizer block 3116 is coupled to MODE_CONTROL block 3118, a Huffman encoder-decoder block 3120, delay block 3126, delay block 3128, and VALUE_REGISTERS block 3130. Input lead 3702 carries the signal difference from MODE_CONTROL block 3118 which determines whether a difference between the new frame and old frame is to be quantized or whether the new frame alone is to be quantized. Values new[x][y] and old[x][y] are supplied on lines 3704 and 3706, respectively, and represent values from memory unit 116 delayed by four clock cycles. Input leads 3708 and 3710 carry the values sign_inv and qindex_inv from the Huffman encoder-decoder block 3120, respectively. Register leads 3318 and 3136 carry signals corresponding to the values qstep and direction from VALUE_REGISTERS block 3130, respectively.

During encoding, quantizer block 3116 performs quantization on the values new[x][y], as dictated by the signal difference and using the values old[x][y], and generates the output values qindex onto output leads 3712, sign onto output lead 3714, and a quantized and then inverse quantized value old[x][y] onto data bus 3102. The quantized and inverse quantized value old[x][y] is written back into memory unit 116.

During decoding, quantizer block 3116 performs inverse quantization on the values old[x][y], as dictated by the signals difference, sign_inv, and qindex_inv, and generates an inverse quantized value, old[x][y], which is supplied to the old portion of memory unit 116 via bus 3102. Lead 3136 carries the value direction supplied by the VALUE_REGISTERS 3130.

The value direction controls whether the quantizer operates in the encoder mode or the decoder mode. Figure 37 illustrates that multiplexers 3716 and 3718 use the direction signal to pass signals corresponding to the appropriate mode (sign and qindex for encoder mode; sign_inv and qindex_inv for decoder mode). Multiplexer 3720 passes either the difference of the new and old data values or passes the new value depending on the value of the difference signal. Absolute value block ABS 3722 converts the value output by multiplexer 3720 to absolute value form and supplies the absolute value form value to block 3724. The output leads of multiplexer 3720 are also coupled to sign block 3726. Sign block 3726 generates a sign signal onto lead 3714 and to multiplexer 3716.

Block 3724 of the quantizer block 3116 is an human visual system (HVS) weighted quantizer having a threshold of qstep. The value on input leads 3728 denoted mag in Figure 37 is quantized via a modulo-qstep division (see Figures 30 and 31 of copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression" and the corresponding text). The resulting quantized index value qindex is output onto leads 3712 to the Huffman encoder block 3120. Multiplexer 3716 receives the sign signal on leads 3714 from block 3726 and also the sign_inv signal on lead 3708. Multiplexer 3716 passes the sign value in the encoder mode and passes the sign_inv value in the decoder mode. Likewise, multiplexer 3718 has as two inputs, the qindex signal on leads 3712 and the qindex_inv signal on leads 3710. Multiplexer 3718 passes the qindex value in the encoder mode and the qindex_inv value in the decoder mode. Inverse quantizer block 3730 inverse quantizes the value output by multiplexer 3718 by the value qstep to generate the value qvalue. Block NEG 3732 reverses the sign of the value on the output lead of block 3730, denoted qvalue in Figure 37. Multiplexer 3734 chooses between the positive and negative versions of qvalue as determined by the signal output from multiplexer 3716.

In the encoder mode, if the difference signal is asserted, then output leads 3712 qindex carry the quantized magnitude of the difference between the new and old data values and the output leads 3736 of multiplexer 3734 carry the inverse quantization of this quantized magnitude of the difference between the new and old values. In the encoder mode, if the difference input is deasserted, then the output leads 3712 qindex carry only the quantized magnitude of the new data value and the value on leads 3736 is the inverse quantization of the quantized magnitude of the new data value.

Adder block 3738 adds the inverse quantized value on leads 3736 to the old[x][y] data value and supplies the result to multiplexer 3740. Accordingly, when the difference signal is asserted, the difference between the old inverse quantized value on leads 3706 and the inverse quantized value produced by inverse quantizer 3730 is determined by adding in block 3738 the opposite of the inverse quantized output of block 3730 to the old inverse quantized value. Multiplexer 3740 passes the output of adder block 3738 back into the OLD portion of memory unit 116 via bus 3102. If, on the other hand, the difference signal is not asserted, then multiplexer 3740 passes the value on leads 3736 to the OLD portion of memory unit 116 via bus 3102. Accordingly, a frame

of inverse quantized values of the most recently encoded frame is maintained in the old portion of memory unit 116 during encoding.

In accordance with one embodiment of the present invention, the value of qstep is chosen so that qstep = 2^n, where $0 \leqq n \leqq 7$, so that quantizer block 3724 and inverse quantizer 3730 perform only shifts by n bits. Block 3724 then becomes in VHDL, where >> denotes a shift to the left, and where mag denotes the value output by block 3722:

```
CASE n is
WHEN 0 => qindex: = mag;
WHEN 1 => qindex: = mag >> 1;
        .
        .
WHEN 7 => qindex: = mag >> 7;
END CASE;
```

Similarly, block 3730 is described in VHDL as follows:

```
CASE n is
WHEN 0 => qvalue : = qindex;
WHEN 1 => qvalue : = (qindex << 1) & "0";
WHEN 2 => qvalue : = (qindex << 2) & "01";
        .
        .
WHEN 7 => qvalue : = (qindex << 7) & "0111111";
```

where << denotes a shift to the right and where & denotes concatenation. The factor concatenated after the shift is 2^{n-1}-1.

The tree processor/encoder-decoder circuit 124 of Figure 31 also includes a MODE_CONTROL block 3118. In the encoder mode, MODE_CONTROL block 3118 determines mode changes as set forth in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression" when trees of data values are traversed to compress the data values into a compressed data stream. In the decoder mode, MODE_CONTROL block 3118 determines mode changes as set forth in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression" when trees of data values are recreated from an incoming compressed data stream of tokens and data values.

MODE_CONTROL block 3118 receives signals from DECIDE block 3112, CONTROL_COUNTER block 3124, TP_ADDR_GEN block 3114, and VALUE_REGISTERS block 3130. MODE_CONTROL block 3118 receives the seven flag values from DECIDE block 3112. The input from CONTROL_COUNTER block 3124 is a four-bit state vector 3138 indicating the state of the CONTROL_COUNTER block 3124. Four bits are needed because the CONTROL_COUNTER block 3124 can be in one of nine states. The input from TP_ADDR_GEN block 3114 is the octave signal carried by leads 3322. The VALUE_REGISTERS block 3130 supplies the values on leads 3316, 3318, 3320, 3132, 3134, and 3136 to MODE_CONTROL block 3118. Additionally, in the decoder mode, buffer 3122 supplies token values which are not Huffman decoded onto leads 3202 and to the MODE_CONTROL block 3118 as shown in Figure 32.

MODE_CONTROL block 3118 outputs a value new_mode which is supplied to TP_ADDR_GEN block 3114 via leads 3402 as well as a token length value T_L which is supplied to buffer block 3122 via leads 3140. In the encoder mode, MODE_CONTROL block 3118 also generates and supplies tokens to buffer block 3122 via leads 3202. Leads 3202 are therefore bidirectional to carry token values from MODE_CONTROL block 3118 to buffer block 3122 in the forward mode, and to carry token values from buffer 3122 to MODE_CONTROL block 3118 in the decoder mode. The token length value T_L, on the other hand, is supplied by MODE_CONTROL block 3118 to buffer block 3122 in both the encoder and decoder modes. MODE_CONTROL block 3118 also generates the difference signal and supplies the difference signal to quantizer block 3116 via lead 3142. MODE_CONTROL block 3118 asserts the difference signal when differences between new and old values are to be quantized and deasserts the difference signal when only new values are to be quantized. Appendix B is a VHDL description of an embodiment of the MODE_CONTROL block 3118 in the VHDL language.

In the encoding process, the MODE_CONTROL block 3118 initially assumes a mode, called pro_mode, from the block immediately below the block presently being encoded in the present tree. For example, the blocks in Figure 36 corresponding to states zz0, .., zz3 in the left-most branch inherit their respective pro_modes from the left-most octave 1 block. Similarly, the left-most octave 1 block in Figure 36 inherits its pro_mode from the root of the tree in octave 2. After the data values of the new and old blocks are read and after the DECIDE block 3112 has generated the flags for the new block as described above, the state machine of

MODE_CONTROL block 3118 determines the new_mode for the new block based on the new data values, the flags, and the pro-mode. The value of new_mode, once determined, is then stored as the current mode of the present block in a mode latch. There is one mode latch for each octave of a tree and one for the low pass data values. The mode latches form a stack pointed to by octave so that the mode latches contain the mode in which each of the blocks of the tree was encoded.

The tree processor circuit of Figures 31 and 32 also comprises a Huffman encoder-decoder block 3120. In the encoder mode, inputs to the Huffman encoder-decoder block 3120 are supplied by quantizer block 3116. These inputs comprise the qindex value and the sign signal and are carried by leads 3712 and 3714, respectively. The outputs of Huffman encoder-decoder 3120 comprise the Huffman encoded value on leads 3142 and the Huffman length H_L on leads 3144, both of which are supplied to buffer block 3122.

In the decoder mode, the input to the Huffman encoder-decoder block 3120 is the Huffman encoded value carried by leads 3204 from buffer block 3122. The outputs of the Huffman encoder-decoder 3120 comprise the Huffman length H_L on leads 3144 and the sign_inv and qindex_inv values supplied to quantizer block 3116 via leads 3708 and 3710, respectively.

The Huffman encoder-decoder block 3120 implements the Huffman table shown in Table 2 using combinatorial logic.

qindex	Huffman code
-38 . . . -512	1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1
-22 . . -37	1 1 0 0 0 0 0 0 1 1 1 1 (|qindex| -22)
-7 . . -21	1 1 0 0 0 0 0 0 (|qindex| -7)
-6	1 1 0 0 0 0 0 1
. . .	. . .
-2	1 1 0 1
-1	1 1 1
0	0
1	1 0 1
2	1 0 0 1
. . .	. . .
6	1 0 0 0 0 0 0 1
7 . . 21	1 0 0 0 0 0 0 0 (|qindex| -7)
22 . . 37	1 0 0 0 0 0 0 0 1 1 1 1 (|qindex| -22)
38 . . 511	1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

Table 2

In the encoder mode, qindex values are converted into corresponding Huffman codes for incorporation into the compressed data stream. Tokens generated by the MODE_CONTROL block 3118, on the other hand, are not encoded but rather are written directly into the compressed data stream.

Figure 38 illustrates one possible embodiment of buffer block 3122 of Figures 31 and 32. The function of buffer block 3122 in the encoder mode is to assemble encoded data values and tokens into a single serial compressed data stream. In the decoder mode, the function of buffer block 3122 is to deassemble a compressed

serial data stream into encoded data values and tokens. Complexity is introduced into buffer block 3122 due to the different lengths of different Huffman encoded data values. As illustrated in Figure 31, buffer 3122 is coupled to FIFO buffer 120 via input-output leads 130, to MODE_CONTROL block 3118 via token value leads 3202 and token length leads 3140, to Huffman encoder-decoder 3120 via leads 3144 and Huffman length leads 3144, to CONTROL_COUNTER 3124 via cycle select leads 3802, and to VALUE_REGISTERS 3130 via leads 3136.

The direction signal carried on leads 3136 from VALUE_REGISTERS block 3130 determines whether the buffer block 3122 operates in the encoder mode or in the decoder mode. In encoder mode, multiplexers 3804, 3806, 3808 and 3814 select the values corresponding to their "E" inputs in Figure 38. In the encoder mode, the buffer block 3122 processes the Huffman encoded value signal present on leads 3142, the token value signal present on leads 3202, the cycle select signal on leads 3802, the Huffman length signal H_L on leads 3144, and the token length signal T_L on leads 3140. The cycle select signal, supplied by CONTROL_COUNTER block 3124 via leads 3802, is supplied to multiplexers 3810 and 3812 to control whether a Huffman encoded value (received from Huffman encoder-decoder block 3120) or whether a non-encoded token value (received from MODE_CONTROL block 3118) is the value presently being assembled into the output data stream.

Figure 39 illustrates a simplified diagram of the buffer block 3122 of Figure 38 when configured in encoder mode. The value s_f is a running modulo sixteen sum of the input token length values and Huffman value length values. The circuit which determines s_f comprises adder block 3902, modulo sixteen divider block 3904, and delay block 3906. When the incoming length value added to the prior value s_f produces a length result of sixteen or greater, block 3904 subtracts sixteen from this length result to determine the new value of s_f. Comparator block 3908 also sends a signal high_low to input lead 3916 of multiplexer 3901 indicating that s_f has exceeded sixteen. Figure 39 shows a barrel shifter 3912 receiving data input values from the output data leads of multiplexer 3901 and from the output data leads of multiplexer 3810. Barrel shifter 3912 sends a 32-bit output signal to a 32-bit buffer 3914. The lower 16-bit output of 32-bit buffer 3914 constitutes the encoded bit stream output of the video encoder/decoder chip which is output onto input/output leads 130.

When the prior value of s_f plus the incoming value length is sixteen or greater, then the lower sixteen bits of buffer 3914 are sent out to FIFO buffer 120 and multiplexer 3901 is set to pass the upper sixteen bits of buffer 3914 back into the lower sixteen bit positions in barrel shifter 3912. The value s_f is then decremented by sixteen. These passed back bits will next become some of the bits in the lower sixteen bits of buffer 3914, on which a subsequent incoming encoded value or token received from multiplier 3810 will be stacked by the barrel shifter starting at location S_f to make sixteen or more packed bits.

Alternatively, if the value of s_f plus the length of the new incoming value is less than sixteen, then multiplexer 3901 is controlled to pass the lower sixteen bits of buffer 3914 back to barrel shifter 3912 and no bits are applied to FIFO buffer 120. The bits of a subsequent incoming encoded value or taken from multiplexer 3810 will be stacked on top of the bits of prior encoded data values or tokens in barrel shifter 3912. Because the value S_f did not exceed sixteen, s_f is not decremented by sixteen.

Figure 40 illustrates a typical output of the barrel shifter 3912 of the buffer 3122 in encoder mode. The maximum length of a Huffman encoded word is sixteen bits. All tokens are two bits in length, where length is the number of bits in the new encoded value or token. The value s in Figure 40 indicates the bit position in the barrel shifter 3912 immediately following the last encoded data value or token present in the barrel shifter. Accordingly, a new encoded value or token is written into barrel shifter 3912 at positions s... s + length. The resulting 32-bit output of the barrel shifter is rewritten to the 32-bit buffer 3914. The comparator block compares the new value of s + length to sixteen. If this value s + length is sixteen or greater as illustrated in Figure 40, then the control signal high_low on multiplexer input lead 3916 is asserted. The lower sixteen bits of the buffer are therefore already completely packed with either bits of data values and/or with bits of tokens. These lower sixteen bits are therefore output to comprise part of the output data stream. The upper sixteen bits, which are incompletely packed with data values and/or tokens, are sent back to the lower sixteen bit positions in the barrel shifter so that the remaining unpacked bits in the lower sixteen bits can be packed with new data bits or new token bits.

If, on the other hand, this value s + length is fifteen or less, then there remain unpacked bits in the lower sixteen bit positions in barrel shifter 3912. These lower bits in barrel shifter 3912 can therefore not yet be output via buffer 3914 onto lines 130. Only when s + length is sixteen or greater will the contents of barrel shifter 3912 be written to buffer 3914 so that the lower sixteen bits will be output via leads 130.

In the decoder mode, buffer 3122 receives an encoded data stream on leads 130, the token length signal T_L on leads 3140 from MODE_CONTROL block 3118, the Huffman encoded length signal H_L on leads 3144, and the control signal cycle select on lead 3802. Multiplexers 3804, 3806, and 3808 are controlled to select values on their respective "D" inputs. Cycle select signal 3802 selects between the Huffman encoded length H_L and the token length T_L depending on whether a data value or a token is being extracted from the in-

coming data stream.

Figure 41 illustrates a simplified diagram of the buffer block 3122 configured in the decoder mode. The value s_i is a running modulo thirty-two sum of the input token length values and Huffman value length values. The circuit which determines the value of s_i comprises adder block 4002, modulo thirty-two divider block 4004, and delay block 4006. When the incoming length value added to s_i results in a value greater than thirty-two, modulo thirty-two divider block 4004 subtracts thirty-two from this value. A comparator block 4008 sends a signal to buffer 3914 indicating when s_i has reached a value greater than or equal to thirty-two. Additionally, comparator block 4008 sends a signal to both buffer 3914 and to multiplexers 3901 and 4010 indicating when s_i has reached a value greater than or equal to sixteen.

Buffer 3122 in the decoder mode also comprises buffer 3914, multiplexers 3901 and 4010, and barrel shifter 4012. In the case of a Huffman encoded data value being the next value in the incoming data stream, sixteen bits of the encoded data stream that are present in barrel shifter 4012 are passed via output leads 3204 to the Huffman decoder block 3120. The number of bits in the sixteen bits that represent an encoded data value depends on the data value itself in accordance with the Huffman code used. In the case of a token being the next value in the incoming data stream, only the two most significant bits from barrel shifter 4012 are used as the token value which is output onto leads 3202 to MODE_CONTROL block 3118. The remaining fourteen bits are not output during this cycle. After a number of bits of either an encoded data value or a two-bit token is output, the value of s_i is updated to point directly to the first bit of the bits in barrel shifter 4012 which follows the bit last output. The circuit comprising adder block 4002, module block 4004, and delay element 4006 adds the length of the previously output value or token to s_i modulo thirty-two to determine the starting location of the next value or token in barrel shifter 4012. Comparator block 4008 evaluates the value of s_i plus the incoming length value, and transmits an active value on lead 4014 when this value is greater than or equal to sixteen and also transmits an active value on lead 4016 if this value is greater than or equal to thirty-two. When s_i is greater or equal to sixteen, the buffer 3914 will read in a new sixteen bits of encoded bit stream bits into its lower half. When $s_i \geq 32$, the buffer 3914 will read a new sixteen bits into its upper half. The two multiplexers 4010 and 3910 following the buffer 3914 rearrange the order of the low and high halves of the buffer 3914 to maintain at the input leads of barrel shifter 4012 the original order of the encoded data stream.

The tree processor/encoder-decoder circuit 124 of Figures 31 and 32 comprises a CONTROL_COUNTER block 3124. CONTROL_COUNTER block 3214 controls overall timing and sequencing of the other blocks of the tree processor/encoder/decoder circuit 124 by outputting the control signals that determine the timing of the operations that these blocks perform. In accordance with one embodiment of the present invention, the tree processor/encoder/decoder 112 is fully pipelined in a nine stage pipeline sequence, each stage occupying one clock cycle. Appendix C illustrates an embodiment of CONTROL_COUNTER block 3124 described in VHDL code.

The signals output by CONTROL_COUNTER block 3124 comprise a read_enable signal on lead 3404, which is active during read cycles, and a write_enable signal on lead 3406, which is active during write cycles. The signals output also comprise memory control signals on leads 3108 and 3110, which control the old and new portions of memory unit 116, respectively, for reading from memory or writing to memory. The signals output also comprise a 4-bit state vector on lead 3138, which supplies MODE_CONTROL block 3118 with the current cycle. The four-bit state vector counts through values 1 through 4 during the "skip" cycle, the value 5 during the "token" cycle, and the values 6-9 during the "data" cycle. The signals output by CONTROL_COUNTER block 3124 also comprise a cycle state value on leads 3802, which signals buffer 3122 when a token cycle or data cycle is taking place.

Figure 42 illustrates a pipelined encoding/decoding process controlled by CONTROL_COUNTER block 3124. Cycles are divided into three types: data cycles - when Huffman encoded/decoded data is being output/input into the encoded bit stream and when old frame values are being written back to memory; token cycles - when a token is being output/input; and skip cycles - the remaining case when no encoded/decoded data is output to or received from the encoded bit stream. A counter in CONTROL_COUNTER block 3124 counts up to 8 then resets to 0. At each sequence of the count, this counter decodes various control signals depending on the current MODE. The pipeline cycles are:

0) read old[0][0] and in encode new[0][0]; skip cycle.
1) read old[1][0] and in encode new[1][0]; skip cycle.
2) read old[0][1] and in encode new[0][1]; skip cycle.
3) read old[1][1] and in encode new[1][1]; skip cycle.
4) DECIDE blocks outputs flags MODE_CONTROL write/read token into/from coded data stream:
generates new_mode, outputs tokens in encode;
generates new_mode, inputs tokens in decode;
token cycle.

5) Huffman encode/decode qindex[0][0], and write old[0][0]; data cycle.
6) Huffman encode/decode qindex[1][0], and write old[1][0]; data cycle.
7) Huffman encode/decode qindex[0][1], and write old[0][1]; data cycle.
8) Huffman encode/decode qindex[1][1], and write old[1][1]; data cycle.

Figure 42 illustrates that once the new_mode is calculated, another block of data values in the tree can be processed. The tree processor/encoder/decoder is thus fully pipelined, and can process four new transformed data values every five clock cycles. To change the pipeline sequence, it is only required that the control signals in the block CONTROL_COUNTER block 3124 be reprogrammed.

ADDITIONAL EMBODIMENTS

In accordance with the above-described embodiments, digital video in 4:1:1 format is output from A/D video decoder 110 on lines 202 to the discrete wavelet transform circuit 122 of video encoder/decoder circuit 112 row by row in raster-scan form. Figure 43 illustrates another embodiment in accordance with the present invention. Analog video is supplied from video source 104 to an A/D video decoder circuit 4300. The A/D video decoder circuit 4300, which may, for example, be manufactured by Philips, outputs digital video in 4:2:2 format on lines 4301 to a horizontal decimeter circuit 4302. For each two data values input to the horizontal decimeter circuit 4302, the horizontal decimeter circuit 4302 performs low pass filtering and outputs one data value. The decimated and low pass filtered output of horizontal decimeter circuit 4302 is supplied to a memory unit 114 such that data values are written into and stored in memory unit 114 as illustrated in Figure 43. The digital video in 4:2:2 format on lines 4301 occurs at a frame rate of 30 frames per second, each frame consisting of two fields. By discarding the odd field, the full 33•3 ms frame period is available for transforming and compressing/decompressing the remaining even field. The even fields are low-pass filtered by the horizontal decimeter circuit 4302 such that the output of horizontal decimeter circuit 4302 occurs at a rate of 30 frames per second, each frame consisting of only one field. Memory unit 114 contains 640 x 240 total image data values. There are 320 x 240 Y data values, as well as 160 x 240 U data values, as well as 160 x 240 V data values.

In order to perform a forward transform, the Y values from memory unit 114 are read by video encoder/decoder chip 112 as described above and are processed by the row convolver and column convolver of the discrete wavelet transform circuit 122 such that a three octave sub-band decomposition of Y values is written into memory unit 116. The three octave sub-band decomposition for the Y values is illustrated in Figure 43 as being written into a Y portion 4303 of the new portion of memory unit 116.

After the three octave sub-band decomposition for the Y values has been written into memory unit 116, the video encoder/decoder chip 112 reads the U image data values from memory unit 114 but bypasses the row convolver. Accordingly, individual columns of U values in memory unit 114 are digitally filtered into low and high pass components by the column convolver. The high pass component G is discarded and the low pass component H is written into U portion 4304 of the new portion of memory unit 116 illustrated in Figure 43. After the U portion 4304 of memory unit 116 has been written with the low pass H component of the U values, video encoder/decoder chip 112 reads these U values from U portion 4304 and processes these U data values using both the row convolver and column convolver of the discrete wavelet transform circuit 122 to perform an additional two octaves of transform to generate a U value sub-band decomposition. The U value sub-band decomposition is stored in U portion 4304 of memory unit 116. Similarly, the V image data values in memory unit 114 are read by video encoder/decoder chip 112 into the column convolver of the discrete wavelet transform circuit 122, the high pass component G being discarded and the low pass component H being written into V portion 4305 of the new portion of memory unit 116. The V data values of V portion 4305 are then read by the video encoder/decoder chip 112 and processed by both the row convolver and the column convolver of discrete wavelet transform circuit 122 to generate a V sub-band decomposition corresponding to the U sub-band decomposition stored in U portion 4304. This process completes a forward three octave discrete wavelet transform comparable to the 4:1:1 three octave discrete wavelet transform described above in connection with Figures 3A-3C. Y portion 4303 of memory unit 116 comprises 320 x 240 data value memory locations; U portion 4304 comprises 160 x 120 data value memory locations; and V portion 4305 comprises 160 x 120 data value memory locations.

The DWT address generator 508 illustrated in Figure 5 generates a sequence of 19-bit addresses on output lines OUT2. In accordance with the presently described embodiment, however, memory unit 114 is a dynamic random access memory (DRAM). This memory unit 114 is loaded from horizontal decimeter circuit 4302 and is either read from and written to by the video encoder/decoder chip 112. For example, in order for the video encoder/decoder chip 112 to access the Y data values in memory unit 114 the inc_R value supplied to DWT address generator 508 by control block 506 is set to 2. This causes the DWT address generator 508 of the video encoder/decoder chip 112 to increment through even addresses as illustrated in Figure 43 such that only

the Y values in memory unit 114 are read. After all the Y values are read from memory unit 114 and are transformed into a Y sub-band decomposition, then base_u_R is changed to 1 and the Channel_start_r is set so that BASE_MUX 3002 of Figure 30 selects the base_u_R to address the first U data value in memory unit 114. Subsequent U data values are accessed because the inc_R value is set to 4 such that only U data values in memory unit 114 are accessed. Similarly, the V data values are accessed by setting the base_v_R value to 3 and setting the Channel_start_r value such that BASE_MUX 3002 selects the base_v_R input leads. Successive V data values are read from memory unit 114 because the inc_R remains at 3.

Because in accordance with this embodiment the video encoder/decoder chip 112 reads memory unit 114, the DWT address generator 508 supplies both read addresses and write addresses to memory unit 114. The read address bus 3018 and the write address bus 3020 of Figure 30 are therefore multiplexed together (not shown) to supply the addresses on the OUT2 output lines of the DWT address generator.

To perform the inverse transform on a three octave sub-band decomposition stored in memory unit 116 of Figure 43, the row and column convolvers of the video encoder/decoder chip 112 require both low and high pass components to perform the inverse transform. When performing the octave 0 inverse transform on the U and V data values of the sub-band decomposition, zeros are inserted when the video encoder/decoder chip 112 is to read high pass transformed data values. In the octave 0 inverse transform, the row convolver is bypassed such that the output of the column convolver is written directly to the appropriate locations in the memory unit 114 for the U and V inverse transform data values. When the Y transform data values in memory unit 116 are to be inverse transformed, on the other hand, both the column convolver and the row convolver of the video encoder/decoder chip 112 are used on each of the three octaves of the inverse transform. The resulting inverse transformed Y data values are written into memory unit 114 in the appropriate locations as indicated in Figure 43.

Figure 44 illustrates a sequence of reading and writing Y data values from the Y portion of the new portion of memory unit 116 in accordance with the embodiment of the present invention illustrated in Figure 1 where memory unit 116 is a static random access memory (SRAM). The dots in Figure 44 represent individual memory locations in a two-dimensional matrix of memory locations adequately wide and deep to store an entire sub-band decomposition of the Y values in a single two-dimensional matrix. The discrete wavelet transform chip 122 reads the memory location indicated R0 during a first time period, outputs a transformed data value during a second time period to the memory location indicated W1, reads another data value from the memory location denoted R2, writes a transformed data value to the memory location denoted W3 and so forth. If memory unit 116 is realized as a dynamic random access memory (DRAM), addressing memory unit 116 in this manner results in a different row of the memory unit being accessed each successive time period. When successive accesses are made to different rows of standard dynamic random access memory, a row address select (RAS) cycle must be performed each time the row address changes. On the other hand, if successive accesses are performed on memory locations that fall in the same row, then only column address select (CAS) cycles need to be performed. Performing a CAS cycle is significantly faster in a standard dynamic random access memory than a RAS cycle. Accordingly, when memory unit 114 is realized as a dynamic random access memory and when memory unit 116 is read and written in the fashion illustrated in Figure 44, memory accesses are slow.

Figure 45 illustrates a sequence of reading and writing memory unit 116 in accordance with another embodiment of the present invention wherein memory unit 116 is realized as a dynamic random access memory. Again, the dots denote individual memory locations and the matrix of memory locations is assumed to be wide enough and deep enough to accommodate the Y portion of the sub-band decomposition in a single two-dimensional matrix. In the first time period, the memory location designated R0 is read. In the next time period, the memory location R1 is read, then R2 is read in a subsequent time period, then R3 is read in a subsequent period, and so forth. In this way one row of low pass component HH values is read into the video encoder/decoder chip 112 using only one RAS cycle and multiple CAS cycles. Then, a second row of low pass component HH data values is read as designated in Figure 45 by numerals R160, R161, R162 and so forth. The last low pass component data value to be read in the second row is designated R319. This row is also read into the video encoder/decoder chip 112 using only one RAS cycle and multiple CAS cycles. Figure 15 illustrates that after reading the data values that the resulting octave 1 transformed data values determined by the discrete wavelet transform chip 122 are now present in the line delays designated 1334 and 1340 illustrated in Figure 13. At this point in this embodiment of the present invention, the row convolver and the column convolver of the discrete wavelet transform chip 122 are stopped by freezing all the control signals except that line delays 1334 and 1340 are read in sequential fashion and written to the Y portion of the new portion of memory unit 116 as illustrated in Figure 45. In this fashion, two rows of memory locations which were previously read in time periods 0 through 319 are now overwritten with the resulting octave 1 transformed values in periods 320 through 639. Only one RAS cycle is required to write the transformed data values in time periods 320 through 479. Similarly, only one RAS cycle is required to write transformed data values during time periods 480 through

639. This results in significantly faster accessing of memory unit 116. Because dynamic random access memory can be used to realize memory unit 116 rather than static random access memory, system cost is reduced considerably.

In accordance with this embodiment of the present invention, the output of the output OUT2 of the column convolver of the video encoder/decoder circuit 112 is coupled to the output leads of block 1332 as illustrated in Figure 13. However, in the forward or inverse transform of any other octave, the output leads OUT2 are coupled to the line delay 1340. Accordingly, in an embodiment in accordance with the memory accessing scheme illustrated in Figure 45, a multiplexer (not shown) is provided to couple either the output of line delay 1340 or the output of adder block 1332 to the output leads OUT2 of the column wavelet transform circuit 704 of Figure 13.

Figure 46 illustrates another embodiment in accordance with the present invention. Memory unit 116 contains a new portion and an old portion. Each of the new and old portions contains a sub-band decomposition. Due to the spatial locality of the wavelet sub-band decomposition, each two-by-two block of low pass component data values has a high pass component consisting of three trees of high frequency two-by-two blocks of data values. For example, in a three octave sub-band decomposition, each two-by-two block of low pass component data values and its associated three trees of high pass component data values forms a 16-by-16 area of memory which is illustrated in Figure 46.

In order for memory unit 116 to be realized in dynamic random access memory (DRAM), the static random access memories (SRAMs) 4600, 4601, 4602 and 4603 which are used as line delays in the discrete wavelet transform circuit 122 are used as cache memory to hold one 16-by-16 block in the new portion of memory unit 116 as well as one 16-by-16 block in the old portion of memory unit 116. This allows each 16-by-16 block of dynamic random access memory realizing the new and old portions of memory unit 116 to be accessed using at most sixteen RAS cycles. This allows the video encoder/decoder chip 112 to use dynamic random access memory for memory unit 116 rather than static random access memory, thereby reducing system cost.

Figure 47 illustrates a time line of a sequence of operations performed by the circuit illustrated in Figure 46. In a first time period, old 16-by-16 block 3 is read into SRAM 1 4601. Because there is only one set of data pins on video encoder/decoder chip 112 for accessing memory unit 116, the 16-by-16 block 0 of the new portion of memory unit 116 is read into SRAM 0 4600 in the second time period. Bidirectional multiplexer 4604 is controlled by select inputs 4605 to couple the 16-by-16 block of old data values now present in SRAM 1 4601 to the bidirectional input port old 4606 of the tree processor/ encoder/decoder circuit 124. Similarly, the 16-by-16 new data values present in SRAM 0 4600 are coupled to the input port new 4607 of the tree processor/encoder/ decoder circuit 124. Accordingly, the tree processor/ encoder/decoder circuit 124 performs tree processing and encoding in a third time period. During the same third time period, the inverse quantized old 16-by-16 block is rewritten into SRAM 1 4601 through multiplexer 4604. In a fourth time period, old 16-by-16 block 2 is read into SRAM 2 4602. Subsequently, in the fifth time period a 16-by-16 block of new data values is read from memory unit 116 into SRAM 0 4600. The new and old 16-by-16 blocks are again provided to the tree processor/encoder/decoder for processing, the inverse quantized 16-by-16 old block being written into SRAM 2 4602. During the period of time when the tree processor/encoder/decoder circuit 124 is performing tree processing and encoding, the inverse quantized 16-by-16 block in SRAM 1 4601 is written back to 16-by-16 block 3 of the old portion of memory unit 116. Subsequently, in the seventh time period, 16-by-16 block 5 of the old portion of memory unit 116 is read into SRAM 1 4601 and in the eighth time period the 16-by-16 block of new data values 4 in memory unit 116 is read into SRAM 0 4600. In the ninth time period, tree processor/encoder/decoder circuit 124 processes the 16-by-16 new and old blocks 4 and 5 while the 16-by-16 block of inverse quantized data values in SRAM 2 4602 is written to 16-by-16 block 2 in the old portion of memory unit 116. This pipelining technique allows the dynamic random access memory (DRAM) to be accessed during each time period by taking advantage of the time period when the tree processor/encoder/decoder circuit 124 is processing and not reading from memory unit 116. Because all accesses of memory unit 116 are directed to 16-by-16 blocks of memory locations, the number of CAS cycles is maximized. Arrows are provided in Figure 4 6 between memory unit 116 and video encoder/decoder circuit 112 to illustrate the accessing of various 16-by-16 blocks of the new and old sub-band decompositions during different time periods. However, because video encoder/decoder chip 112 only has one set of data leads through which data values can be read from and written to memory unit 116, the input/output ports on the right sides of dual port static random access memories 4600-4602 are bussed together and coupled to the input/output data pins of the video encoder/decoder chip 112.

In order to avoid the necessity of providing an additional memory to realize first-in-first-out (FIFO) memory 120, SRAM 3 4603, which is used as a line delay in the column convolver of the video encoder/decoder chip 112, is coupled to the tree processor/encoder/decoder circuit 124 to buffer the compressed data stream for encoding and decoding operations between the ISA bus 106 and the video encoder/decoder chip 112. This

sharing of SRAM 3 is possible because the discrete wavelet transform circuit 122 operates in a first time period and the tree processor/encoder-decoder circuit 124 operates in a second time period.

When the tree processor/encoder/decoder circuit 124 is performing the decoding function, the new portion of memory unit 116 is not required and SRAM 0 is unused. The read 0, read 1, and read 4 time periods of the time line illustrated in Figure 47 are therefore omitted during decoding.

Although the present invention has been described by way of the above described specific embodiments, the invention is not limited thereto. Adaptations, modifications, rearrangements and combinations of various features of the specific embodiments may be practiced without departing from the scope of the invention. For example, an integrated circuit chip may be realized which performs compression but not decompression and another integrated circuit chip may be realized which performs decompression but not compression. Any level of integration may be practiced including placing memory units on the same chip with a discrete wavelet transform circuit and a tree processor/encoder-decoder circuit. The invention may be incorporated into consumer items including personal computers, video cassette recorders (VCRs), video cameras, televisions, compact disc (CD) players and/or recorders, and digital tape equipment. The invention may process still image data, video data and/or audio data. Filters other than four coefficient quasi-Daubechies forward transform filters and corresponding four coefficient reconstruction (inverse transform) filters may be used including filters disclosed in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression". Various start and end forward transform filters and various corresponding start and end reconstruction (inverse transform) filters may also be used including filters disclosed in copending Patent Cooperation Treaty (PCT) application filed March 30, 1994, entitled "Data Compression and Decompression". Tokens may be encoded or unencoded. Other types of tokens for encoding other information including motion in consecutive video frames may be used. Other types of encoding other than Huffman encoding may be used and different quantization schemes may be employed. The above description of the preferred embodiments is therefore presented merely for illustrative instructional purposes and is not intended to limit the scope of the invention as set forth in the appended claims.

APPENDIX A: VHDL Language Implementation of CONTROL_ENABLE Block 3420

```
--The state machine to control the address counters#
--works for 3 octave decomposition in y & 2 in u|v#

use work.DWT_TYPES.all;
use work.dff_package.all;

entity U_CONTROL_ENABLE is

port(
ck : in bit ;
reset : in t_reset ;
new_channel,channel : in t_channel ;
c_blk : in BIT_VECTOR(1 to 3) ;
subband : in BIT_VECTOR(1 to 2) ;
load_channel : in     t_load ;
new_mode : in  t_mode ;

out_1 : out BIT_VECTOR(1 to 3);
out_2 : out t_octave;
out_3 : out bit;
out_4 : out bit;
out_5 : out t_state) ;

end U_CONTROL_ENABLE;

architecture behave OF U_CONTROL_ENABLE IS
signal            state:t_state;
signal            new_state_sig:t_state;
BEGIN

state_machine:PROCESS(reset,new_channel,channel,c_blk,subband,load_channel,new_mode,state,new_state_sig)

VARIABLE en_blk:BIT_VECTOR(1 to 3) := B"000";
--enable blk_count#
```

```
variable            lpf_block_done:bit := '0';
--enable x_count for LPF#
variable            tree_done:bit := '0';
--enable x_count for other subbands#
variable          reset_state:t_state;
variable             new_state:t_state ;
variable              octave:t_octave := 0;
--current octave#
variable          start_state:t_state;
-- dummy signals for DFI

BEGIN
-- default initial conditions
      en_blk:=b"000";
      lpf_block_done:= '0';
      tree_done:= '0';
      octave:= 0;
      reset_state:=up0;
      new_state:=state;
      start_state:=up0;
--set up initial state thro mux on reset, on HH stay in zz0 state#
CASE channel  IS
WHEN                    u|v   =>  start_state:=  down1;
WHEN                    y     =>  start_state:=  up0;
END CASE;
CASE reset  IS
WHEN   rst  =>  reset_state:=   start_state;
WHEN  OTHERS      =>  reset_state := state;
END CASE;

CASE  reset_state  IS
WHEN     up0      =>   octave :=2;
                       en_blk(3):= '1';
                       CASE c_blk(3)  IS
                       WHEN  '1'   =>     CASE subband  IS
                                          WHEN     B"00"     =>   lpf_block_done := '1';
--clock x_count for LPF y channel#
```

```
                                        WHEN        OTHERS  =>  new_state := up1;
--change state when count done#
                                        END CASE;

                                        CASE new_mode              IS
--in luminance & done with that tree#
                                        WHEN      stop      =>   tree_done := '1';
                                        WHEN        OTHERS  =>  null;
                                        END CASE;
                        WHEN            OTHERS  =>  null;
                        END CASE;
WHEN      up1       =>  octave :=1;
                        en_blk(2):= '1';
                        CASE c_blk(2)  IS
                        WHEN      '1'       =>   new_state := zz0;

                                        CASE new_mode        IS
--in luminance, terminate branch & move to next branch#
                                        WHEN      stop      =>   new_state := down1;
                                             en_blk(3):= '1';
                                        WHEN        OTHERS  =>  null;
                                        END CASE;
                        WHEN            OTHERS  =>  null;
                        END CASE;
WHEN      zz0       =>  octave :=0;
                        en_blk(1):= '1';
                        CASE c_blk(1)  IS
                        WHEN      '1'       =>   new_state := zz1;
                                        en_blk(2):= '1';
                        WHEN            OTHERS  =>  null;
                        END CASE;
WHEN      zz1       =>  octave :=0;
                        en_blk(1):= '1';
                        CASE c_blk(1)  IS
                        WHEN      '1'       =>   new_state := zz2;
                                        en_blk(2):= '1';
                        WHEN            OTHERS  =>  null;
```


EP 0 622 741 A2

```
                    END CASE;
WHEN    zz2     =>  octave :=0;
                    en_blk(1):= '1';
                    CASE c_blk(1)  IS
                    WHEN    '1'     =>  new_state := zz3;
                                        en_blk(2):= '1';
                    WHEN            OTHERS  =>  null;
                    END CASE;
WHEN    zz3     =>  octave :=0;
                    en_blk(1):= '1';
--now decide the next state, on block(1) carry check the other block carries#
                    --nowdecide the next state, on block(1) carry check the other block carries
--
                    CASE c_blk(1)  IS
                    WHEN    '1'     =>  new_state := down1;
                                        en_blk(2):= '1';
--roll over to 0#
                                        en_blk(3):= '1' ;
--because state zz3 clock 1 pulse#
                    WHEN            OTHERS  =>  null;
                    END CASE;

WHEN    down1   =>  octave :=1;
                    en_blk(2):= '1';
                    CASE c_blk(2)  IS
                    WHEN  '1'   =>      CASE subband  IS
                                        WHEN    B"00"   =>  lpf_block_done := '1' ;
--clock x_count for LPP u|v channel#
                                        WHEN        OTHERS  =>  new_state := zz0 ;
--change state when count done#
                                        END CASE;

                                        CASE new_mode IS
                                        WHEN  stop => CASE channel IS
--stop so finish thisbranch & move on#
                                                      WHEN    u|v     =>  tree_done := '1';
```

```
                                                    WHEN      y  =>   en_blk(3) := '1';
                                                               CASE c_blk(3)       IS
--move to next tree#
                                                               WHEN      '1'       =>   tree_done := '1';
                                                               WHEN        OTHERS  =>  new_state := down1;
                                                               END CASE;

                                                    WHEN       OTHERS  =>  null;
                                                    END CASE;
                                          WHEN      OTHERS  =>  null;
                                          END CASE;
                                WHEN      OTHERS  =>  null;
                                END CASE;

END CASE;

CASE channel  IS
WHEN  u|v   =>      IF c_blk(1)='1' AND c_blk(2)= '1'  THEN tree_done := '1';
                    ELSE null;
                    END IF;

WHEN  y     =>      IF c_blk(1)='1' AND c_blk(2)='1' AND c_blk(3)= '1'  THEN
                         tree_done := '1';
                    ELSE  null;
                    END IF;
END CASE;

--now change to start state if the sequence has finished#
CASE  tree_done     IS
--in LPF state doesnt change when block done#
WHEN      '1'       =>   new_state := start_state;
WHEN        OTHERS  =>  null;
END CASE;
--on channel change, use starting state for new channel#
CASE    load_channel         IS
--in LPF state doesnt change when block done#
WHEN  write =>    CASE new_channel  IS
```

```
                WHEN  y => new_state:= up0;
                WHEN  u|v =>  new_state:=down1;
                END CASE;
            WHEN        OTHERS  =>  null;
            END CASE;

            new_state_sig<=new_state;

            out_1 <= en_blk;
            out_2 <= octave;
            out_3 <= tree_done;
            out_4 <= lpf_block_done;
            out_5 <=reset_state;

            END PROCESS;

            DF1(ck,new_state_sig,state);
            END behave;

            CONFIGURATION CONTROL_ENABLE_CON OF U_CONTROL_ENABLE is
            FOR behave
            END FOR;
            END  CONTROL_ENABLE_CON;
```

APPENDIX B: VHDL Language Implementation of MODE_CONTROL Block 3118

```
--generates the new_mode from the old, and outputs control signals to the tokeniser--

use work.DWT_TYPES.all;
use work.dff_package.all;


entity U_MODE_CONTROL  IS
PORT(
  ck : in bit ;
  reset : in  t_reset ;
  intra_inter : in  t_intra ;
  lpf_done : in bit ;
  flags : in BIT_VECTOR(1 to 7);
  token_in : in   BIT_VECTOR(1 to 2) ;
  octave : in t_octave ;
  state : in t_state ;
  direction : in t_direction ;
  load_mode_in : in t_load ;
  cycle: in t_cycle ;

out_1:out t_mode;
out_2:out t_mode;
out_3:out BIT_VECTOR(1 to 2) ;
out_4:out t_diff;
out_5:out BIT_VECTOR(1 to 2) ;
out_6:out t_mode));



end U_MODE_CONTROL;



architecture behave OF U_MODE_CONTROL IS
```

```
--new_mode,proposed mode,current token,difference,token_length, --
signal nzflag:bit;
signal origin:bit;
signal noflag:bit;
signal ozflag:bit;
signal motion:bit;
signal pro_new_z:bit;
signal pro_no_z:bit;
signal lpf_done_del:bit;
signal load_mode:t_load_vec(1 to 4);
signal load_next:t_load;
signal pre_mode_sig:t_mode;
signal pro_mode_sig:t_mode;
signal new_mode_sig:t_mode;
signal mode:t_mode;
signal diff_sig:t_diff;
signal diff_out:t_diff;
signal mode_regs:t_mode_vec(1 to 4);
BEGIN

      nzflag <= flags(1);
      origin <= flags(2);
      noflag <= flags(3);
      ozflag <= flags(4);
      motion <= flags(5);
      pro_new_z  <=  flags(6);
      pro_no_z  <=  flags(7);

      DF1(ck,lpf_done,lpf_done_del);          --synchronise mode change at end of LPP--

--the proposed value for the mode at that octave, flags etc will change this value as necessary--
--proposed, or inherited mode from previous tree--

MODE_CONTROL:PROCESS( nzflag,origin,noflag,ozflag,motion,pro_new_z,pro_no_z,lpf_done_del,token_in,direction,
                      mode_regs ,state,reset,intra_inter,octave)

variable          pro_mode :t_mode;
```

```
variable          new_mode :t_mode;
variable        token_out :bit_vector(1 to 2);
variable        difference :t_diff;
variable        token_length :bit_vector(1 to 2);
variable        pro_flag :bit;

BEGIN

--initialise variables

      CASE reset  IS
      WHEN  rst   =>     CASE intra_inter   IS
--reset on frame start, so do lpf--
                              WHEN  intra => pro_mode:= lpf_still;
                              WHEN        OTHERS  =>   pro_mode:=lpf_send;
                              END CASE;
      WHEN      OTHERS  =>      CASE  lpf_done_del  IS
                                      WHEN  '1'   =>    CASE intra_inter   IS
--store default mode in mode(4)--
                                                              WHEN   intra =>  pro_mode:=still;
                                                              WHEN        OTHERS  =>  pro_mode:= send;
                                                              END CASE;

                                      WHEN      OTHERS  =>      CASE  state  IS
mode_regs(3);                                                                 WHEN down1 => pro_mode:=
--jump sideways in oct 1--

mode_regs(4);                                                                 WHEN up0 =>  pro_mode:=

octave  IS                                                                    WHEN      OTHERS  =>   CASE

WHEN 0 =>pro_mode:= mode_regs(1);

WHEN 1 =>pro_mode:= mode_regs(2);

WHEN 2 =>pro_mode:= mode_regs(3);
```

```
WHEN 3 =>pro_mode:= mode_regs(4);

CASE;                                                                    END

                                                                  END CASE;
                                        END CASE;
      END CASE;

                new_mode := pro_mode;          --inherit the previous mode--
                token_out := B"00";
                difference := nodiff;
                token_length := B"00";
                pro_flag := '0';

CASE direction  IS
WHEN  forward =>

                      CASE  pro_mode  IS

                      WHEN lpf_stop|stop => null;
                      WHEN  void  =>     CASE ozflag  IS
                                               WHEN    '1' =>   new_mode := stop;
                                               WHEN     OTHERS  =>  null;
                                               END CASE;

                      WHEN  void_still =>  null;
--intra so must zero out all of tree--

                      WHEN     still_send     =>   token_length := B"01";
                                                       IF nzflag='1' OR pro_new_z ='1'  THEN token_out :=
B"00";

                                                                CASE ozflag  IS
                                                                WHEN    '1' =>   new_mode
:= stop;
                                                                WHEN     OTHERS  =>
new_mode := void;
                                                                END CASE;
```


EP 0 622 741 A2

```
                                                    ELSE   token_out := B"10";
                                                           new_mode:= still_send;
                                                    END IF;

                    WHEN  send  =>    CASE ozflag  IS
                                                          WHEN   '1'  =>   token_length := B"01";

token_out := B"00";                                       IF nzflag = '1' OR pro_new_z='1'  THEN

new_mode:= stop;

                                                          ELSE                    token_out := B"10";
still_send;                                                                            new_mode:=

                                                          END IF;

                                                          WHEN                    OTHERS  =>  token_length := B"10";

NOT(nzflag) ='1'                                          IF(NOT(noflag) ='1' OR motion = '1') AND

                                                          THEN

pro_new_z;                                                                  CASE origin  IS
                                                                            WHEN     '1'=>   pro_flag :=
pro_no_z;                                                                   WHEN  OTHERS => pro_flag :=

difference:= diff;



                                                                            END CASE;

:= B"10";                                                                   CASE pro_flag  IS
                                                                            WHEN     '1' =>   token_out
new_mode:= void;

origin  IS                                                                  WHEN     OTHERS  =>  CASE
```

```
WHEN '1' =>  token_out := B"01";
new_mode:= still_send;
WHEN  OTHERS  =>  token_out := B"11";
new_mode:= send;
CASE;
                                                                    END
                                                          END CASE;
                                                ELSE
='1')AND nzflag ='1'                                     IF (motion  ='1' OR origin
                                                          THEN
:= B"10";                                                       token_out
new_mode:= void;
B"00";                                                    ELSE        token_out :=
new_mode:= stop;
                                                          END IF;
                                                END IF;
                                          END CASE;
                        WHEN      still     =>   token_length := B"01";
                                                 IF nzflag ='1' OR pro_new_z = '1'
                                                 THEN  token_out := B"00";
--zero out tree--                                       new_mode:= void_still;
                                            ELSE   token_out := B"10";
```

```
                                                    new_mode:= still;
                                                END IF;

                    WHEN    lpf_still       =>  token_out := B"00";
                                                    token_length:= B"00";

                    WHEN    lpf_send =>  difference := diff;
                                                    token_length:= B"01";

                                                IF noflag ='1' OR pro_no_z = '1'
                                                THEN        token_out := B"00";
                                                                new_mode:= lpf_stop;
                                            ELSE  token_out := B"10";
--as mode stop but for this block only--                        new_mode:= lpf_send;

                                                END IF;
                    END CASE;

    WHEN  inverse =>
                    CASE  pro_mode  IS
                    WHEN lpf_stop|stop => null;

                    WHEN  void  =>     CASE ozflag  IS
                                                WHEN  '1'    =>   new_mode := stop;
                                                WHEN         OTHERS  =>  null;
                                                END CASE;

                    WHEN  void_still  => null;

                    WHEN  send  =>     CASE ozflag  IS
                                                WHEN    '1' =>   token_length := B"01";
--repeat of still-send code--

                                                                CASE token_in(1)  IS
still_send;                                                     WHEN    '1'  =>   new_mode :=

                                                                WHEN    '0'  =>   new_mode :=
stop;
```

```
                                                    END CASE;

                                          WHEN   OTHERS  =>  token_length := B"10";
                                                             CASE token_in  IS
                                                             WHEN     B"11"     =>   difference
:= diff;

new_mode:= send;

                                                             WHEN     B"01"     =>   new_mode
:= still_send;

                                                             WHEN     B"10"     =>   new_mode
:= void;

                                                             WHEN     B"00"     =>   new_mode
:= stop;

                                                             END CASE;

                                    END CASE;

                    WHEN     still_send     =>   token_length := B"01";
                                                 CASE token_in(1)  IS
                                                 WHEN     '1'      =>   new_mode := still_send;
                                                 WHEN  '0'   =>         CASE ozflag  IS
                                                                        WHEN    '1' =>
new_mode := stop;
                                                                        WHEN        OTHERS
=>  new_mode := void;
                                                                        END CASE;
                                                 END CASE;

                    WHEN     still    =>   token_length := B"01";
                                           CASE token_in(1)  IS
                                           WHEN     '1'      =>   new_mode := still;
                                           WHEN     '0'      =>   new_mode := void_still;
                                           END CASE;

                    WHEN     lpf_send =>   difference := diff;
                                           token_length:= B"01";
```


EP 0 622 741 A2

```
                                        CASE token_in(1)      IS
                                        WHEN      '0'         =>   new_mode := lpf_stop;
                                        WHEN      '1'         =>   new_mode := lpf_send;
                                        END CASE;

                    WHEN  lpf_still    => null;
                    END CASE;

END CASE;

--relate variable to corresponding signals

out_2 <= pro_mode;
pro_mode_sig <= pro_mode;
out_3 <= token_out;
out_5 <= token_length;
out_6 <= new_mode;
new_mode_sig <= new_mode;
diff_sig <= difference;


END PROCESS MODE_CONTROL;

out_1 <=  mode;
out_4 <= diff_out;

pre_mode_sig <= pro_mode_sig WHEN reset = rst OR lpf_done_del= '1' ELSE
                        mode;

--save the new mode& difference during a token cycle, when the flags and tokens are valid--
-- on lpf_still & inverse no token cycles so load on skip cycle, just so next_mode is defined

load_next <= write WHEN cycle = token_cycle ELSE
                 write WHEN cycle = skip_cycle AND pro_mode_sig=lpf_still AND direction = inverse ELSE
                 read;
```


EP 0 622 741 A2

```
DFF_INIT(ck,no_rst,load_next,new_mode_sig,mode);
DFF_INIT(ck,no_rst,load_next,diff_sig,diff_out);

--now write the new mode value into the mode stack at end of cycle, for later use --
--dont update modes at tree base from lpf data, on reset next(1) is undefined--
--store base mode in mode(3)& mode(4), base changes after lpf--

load_mode  <=   (read,read,write,write)  WHEN   reset=rst OR lpf_done_del= '1' ELSE
                (write,write,read,read)  WHEN  octave= 1 AND load_mode_in= write ELSE
                (read,write,write,read)  WHEN   octave = 2 AND load_mode_in=write ELSE
                (read,read,read,read)    ;

DFF_INIT(ck,no_rst,load_mode(1),pre_mode_sig,mode_regs(1));
DFF_INIT(ck,no_rst,load_mode(2),pre_mode_sig,mode_regs(2));
DFF_INIT(ck,no_rst,load_mode(3),pre_mode_sig,mode_regs(3));
DFF_INIT(ck,no_rst,load_mode(4),pre_mode_sig,mode_regs(4));

END behave;

CONFIGURATION MODE_CONTROL_CON OF U_MODE_CONTROL is
FOR behave
END FOR;
END  MODE_CONTROL_CON;
```


EP 0 622 741 A2

APPENDIX C: VHDL Language Implementation of CONTROL_COUNTER 3124

```
--a counter to control the sequencing ofw, token, huffman cycles--
--decide reset is enabled 1 cycle early, and latched to avoid glitches--
--lpf_stop is a is a dummy mode to disable the block writes&huffman data--
--cycles for that block--
use work.DWT_TYPES.all;
use work.dff_package.all;

entity U_CONTROL_COUNTER  IS
PORT(
  ck : in bit ;
  reset : in t_reset ;
  mode,new_mode : in t_mode ;
  direction  : in t_direction ;

out_0 : out t_load;
out_1 : out t_cycle;
out_2 : out t_reset;
out_3 : out bit;
out_4 : out bit;
out_5 : out t_load;
out_6 : out t_cs;
out_7 : out t_load;
out_8 : out t_cs) ;

--mode load,cycle,decide reset,read_addr_enable,write_addr_enable,load flags--
--decode write_addr_enable early and latch to avoid feedback loop with pro_mode--
--in MODE_CONTROL--
end U_CONTROL_COUNTER;


architecture behave OF U_CONTROL_COUNTER IS
COMPONENT COUNT_SYNC
GENERIC (n:integer);
PORT(
```

```
    ck:in bit ;
    reset:in t_reset;
    en:in bit;
    q:out bit_vector(1 to n);
    carry:out bit);
    end COMPONENT;

    signal write_del:bit;
    signal write_sig:bit;
    signal decide_del:t_reset;
    signal decide_sig:t_reset;
    signal count_reset:t_reset;
    signal count_len:t_length;
    signal count_1:BIT_VECTOR(1 to 4);
    signal count_2:bit;
    signal always_one:bit:='1';
    BEGIN

    count_len  <=  U_TO_I( count_1);

    control:PROCESS(ck,count_reset,direction,mode,new_mode,count_len)

        VARIABLE        cycle : t_cycle;
        VARIABLE        decide_reset :  t_reset;
        VARIABLE        load_mode : t_load;
        VARIABLE        load_flags : t_load;
        VARIABLE        cs_new : t_cs;
        VARIABLE        cs_old : t_cs;
        VARIABLE        rw_old : t_load;
        VARIABLE        read_addr_enable : bit;
        VARIABLE        write_addr_enable : bit;

    BEGIN
            cycle := skip_cycle;
        decide_reset :=  no_rst;
        load_mode := read;
        load_flags := read;
```

```
        cs_new := no_sel;
        cs_old := sel;
        rw_old := read;
        read_addr_enable := '0';
        write_addr_enable := '0';

    CASE direction  IS
    WHEN forward =>     CASE mode  IS
                            WHEN send|still_send|lpf_send =>     CASE count_len  IS
                                                                WHEN    0 to  3 => read_addr_enable := '1';
                                                                                   cs_new:= sel;
                                                                WHEN    4 =>  cycle := token_cycle;
                                                                                   load_flags:= write;
                                                                                   write_addr_enable:= '1';

                                                                WHEN    5 to  7 =>     write_addr_enable:= '1';
                                                                                   CASE new_mode  IS
    cycle := skip_cycle;                                                           WHEN stop|lpf_stop =>

    rw_old:= read;

    cs_old:= no_sel;

    skip_cycle;                                                                    WHEN void =>  cycle :=

    rw_old:= write;

    data_cycle;                                                                    WHEN  OTHERS  => cycle :=

    rw_old:= write;

                                                                                   END CASE;
                                                                WHEN    8 =>  decide_reset := rst;
                                                                                   CASE new_mode  IS
    cycle := skip_cycle;                                                           WHEN stop|lpf_stop =>
```

```
                    WHEN void =>  cycle :=
                                                    WHEN  OTHERS  => cycle :=
                                                END CASE;
                                            WHEN  OTHERS  => null;
                                            END CASE;
                                    WHEN still =>   CASE count_len  IS
                                                        WHEN      0 to  3 => read_addr_enable := '1';
                                                                             cs_new:= sel;
                                                        WHEN      4 =>  cycle := token_cycle;
                                                                        write_addr_enable := '1';
                                                                        load_flags:= write;
                                                        WHEN      5 to  7 => rw_old := write;
                                                                             write_addr_enable := '1';
                                                                             CASE new_mode  IS
                                                                             WHEN void_still =>  cycle
                                                        WHEN  OTHERS  => cycle :=
                                                        END CASE;
                                        WHEN      8 =>  decide_reset := rst;
rw_old:= read;
cs_old:= no_sel;
skip_cycle;
load_mode:= write;
rw_old:= write;
data_cycle;
load_mode:= write;
rw_old:= write;
:= skip_cycle;
data_cycle;
```

```
                                                    rw_old:= write;
                                                    load_mode:= write;
                                                    CASE new_mode  IS
:= skip_cycle;                                      WHEN void_still =>  cycle

data_cycle;                                         WHEN  OTHERS  => cycle :=

                                                    END CASE;

                                        WHEN  OTHERS  => null;
                                        END CASE;

                            WHEN lpf_still =>   CASE count_len  IS
                                        WHEN 0 to  3 => read_addr_enable := '1';
                                                    cs_new:= sel;
                                        WHEN     4 =>  cycle := token_cycle;
                                                    write_addr_enable := '1';
                                                    load_flags:= write;
                                        WHEN     5 to  7 => cycle := data_cycle;
                                                    rw_old:= write;
                                                    write_addr_enable := '1';
                                        WHEN     8 =>  cycle := data_cycle;
                                                    rw_old:= write;
                                                    decide_reset:= rst;
                                                    load_mode:= write;
                                        WHEN  OTHERS  => null;
                                        END CASE;

                            WHEN void =>        CASE count_len  IS
                                        WHEN 0 to 3 => read_addr_enable := '1';
                                                    cs_new:= sel;
                                        WHEN     4 =>  load_flags := write;
                                                    cycle:= token_cycle;

--dummy token cycle for mode update--                write_addr_enable := '1';
                                        WHEN     5 to 7 => write_addr_enable := '1';
--keep counters going--
```

```
                                CASE new_mode  IS
                                WHEN stop =>  rw_old :=
read;
ce_old:= no_sel;
write;
                                              WHEN  OTHERS  => rw_old :=
                                                END CASE;
                                WHEN    8 =>  decide_reset := rst;
                                                CASE new_mode  IS
                                                WHEN stop =>  rw_old :=
read;
ce_old:= no_sel;
:= write;
rw_old:= write;
                                                WHEN  OTHERS  => load_mode
                                                END CASE;
                                WHEN  OTHERS  => null;
                                END CASE;
                        WHEN void_still =>
                                CASE count_len  IS
                                WHEN 0 =>  write_addr_enable := '1';
--allow for delay--
                                WHEN    1 to  3 => write_addr_enable := '1';
                                                   rw_old:= write;
                                WHEN    4 =>  rw_old := write;
                                                   load_mode:= write;
                                                   decide_reset:= rst;
                                WHEN  OTHERS  => null;
                                END CASE;
                        WHEN  OTHERS  => null;
                        END CASE;
                WHEN inverse =>     CASE mode  IS
```

```
                    WHEN send|still_send|lpf_send =>      CASE count_len  IS
                                                    WHEN 0 to 3 => read_addr_enable := '1';
                                                    WHEN     4 =>  cycle := token_cycle;
                                                                   write_addr_enable := '1';
                                                                   load_flags:= write;
                                                    WHEN     5 to 7 => write_addr_enable := '1';
                                                                   CASE new_mode  IS
cycle := skip_cycle;                                               WHEN stop|lpf_stop =>

rw_old:= read;

cs_old:= no_sel;

skip_cycle;                                                        WHEN void =>  cycle :=

rw_old:= write;

data_cycle;                                                        WHEN  OTHERS  => cycle :=

rw_old:= write;

                                                                   END CASE;
                                                    WHEN     8 =>  decide_reset := rst;
                                                                   CASE new_mode  IS
cycle := skip_cycle;                                               WHEN stop|lpf_stop =>

rw_old:= read;

cs_old:= no_sel;

skip_cycle;                                                        WHEN void =>  cycle :=

load_mode:= write;

rw_old:= write;
```

```
data_cycle;                                                    WHEN  OTHERS  => cycle :=
load_mode:= write;
rw_old:= write;
                                                               END CASE;
                                        WHEN  OTHERS  => null;
                                        END CASE;
                          WHEN still =>  CASE count_len  IS
                                        WHEN        0 => null ;
--skip to allow reset in huffman--
                                        WHEN     1 =>  cycle := token_cycle;
                                                       write_addr_enable := '1';
                                        WHEN     2 to  4 => rw_old := write;
                                                       write_addr_enable := '1';
                                                       CASE new_mode  IS
:= skip_cycle;                                         WHEN void_still =>  cycle
data_cycle;                                            WHEN  OTHERS  => cycle :=
                                                       END CASE;
                                        WHEN  5     => rw_old:=write;
                                                       decide_reset:= ret;
                                                       load_mode:= write;
                                                       CASE new_mode  IS
:= skip_cycle;                                         WHEN void_still =>  cycle
data_cycle;                                            WHEN  OTHERS  => cycle :=
                                                       END CASE;
                                        WHEN  OTHERS  => null;
                                        END CASE;
                          WHEN lpf_still =>  CASE count_len  IS
                                        WHEN              0 =>null ;
```

```
--match with previous--

--skip for write enb delay--

                                WHEN    1 =>  write_addr_enable := '1';

                                WHEN    2 to  4 => cycle := data_cycle;
                                                   rw_old:= write;
                                                   write_addr_enable := '1';
                                WHEN    5 =>  cycle := data_cycle;
                                                   rw_old:= write;
                                                   decide_reset:= rst;
                                                   load_mode:= write;
                                WHEN  OTHERS  => null;
                                END CASE;
                        CASE count_len  IS
                                WHEN 0 to 3 => read_addr_enable := '1';
                                WHEN      4 => load_flag := write;
                                                   cycle:= token_cycle;

                WHEN void =>

--dummy token cycle for mode update--

                                WHEN    5 to 7 => write_addr_enable := '1';
                                                  CASE new_mode  IS
                                                  WHEN stop =>  rw_old :=

        read;

cs_old:= no_sel;

write;

                                                  WHEN  OTHERS  => rw_old :=

                                WHEN    8 =>  decide_reset := rst;
                                                  END CASE;
                                                  CASE new_mode  IS
                                                  WHEN stop =>  rw_old :=

read;

cs_old:= no_sel;

:= write;

                                                  WHEN  OTHERS  => load_mode

rw_old:= write;
```

```
                                        END CASE;
                                    WHEN  OTHERS  => null;
                                    END CASE;
                                CASE count_len  IS
                                    WHEN        0     =>null ;

                                    WHEN      1 =>  write_addr_enable := '1';

                                    WHEN      2 to  4 => write_addr_enable := '1';
                                                         rw_old:= write;
                                    WHEN      5 =>  rw_old := write;
                                                         load_mode:= write;
                                                         decide_reset:= rst;
                                    WHEN  OTHERS  => null;
                                    END CASE;
                                WHEN void_still =>
--match with reset--

--dummy as write delayed--




                                WHEN  OTHERS  => null;
                                END CASE;
      END CASE;

write_sig <=write_addr_enable;
decide_sig <= decide_reset;

DFF(ck,reset,write_sig,write_del);
out_0 <= load_mode ;
out_1 <= cycle;
out_2 <= decide_sig;
out_3 <= read_addr_enable;
out_4 <= write_del;
0out_5 <= load_flags;
out_6 <= cs_new;
out_7 <=  rw_old;
out_8 <=  cs_old;

END PROCESS;

WITH reset SELECT
count_reset <= rst WHEN  rst,
```

```
                    decide_sig WHEN OTHERS;
control_cnt: count_sync GENERIC MAP(4) PORT MAP(ck,count_reset,always_one,count_1,count_2);

END behave;

CONFIGURATION CONTROL_COUNTER_CON OF U_CONTROL_COUNTER is
FOR behave
    FOR ALL:count_sync  USE ENTITY WORK.count_sync(behave);
     END FOR;
END FOR;
END  CONTROL_COUNTER_CON;
```

APPENDIX D: VHDL Language Implementation of Video Encoder/Decoder Integrated Circuit Chip

```
--VHDL Description of Discrete Wavelet Transform Circuit--
--the string base address calculators--
use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;
entity U_NOMULT  IS
PORT(
  ck : in bit ;
  reset : in t_reset ;
  col_end : in bit ;
  mux_control : in t_mux4 ;
  incr : in t_memory_addr;
  oct_add_factor : in   t_memory_addr ;
  base_u,base_v : in  BIT_VECTOR(1 to 19);

out_1 : out t_memory_addr);
end U_NOMULT;


architecture behave OF U_NOMULT IS

signal mux:t_memory_addr;
signal next_addr:t_memory_addr;
signal dff_out:t_memory_addr;
signal add:t_memory_addr;

BEGIN
WITH mux_control SELECT
next_addr  <=  add
                        0                       WHEN uno,
                        U_TO_I(base_u)          WHEN dos,
                        U_TO_I(base_v)          WHEN tres,
                                                WHEN quatro;
```

```
BEGIN
temp <=       '0' WHEN reset=rst ELSE
             '1' WHEN j='1' ELSE
              q ;

DFF(ck,temp,q);
out_1 <= q;
end behave;

CONFIGURATION JKFF_CON OF JKFF       is
FOR behave
END FOR;
END JKFF_CON;

use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;

entity TOGGLE IS
PORT(
 ck : in bit ;
reset : in t_reset ;
j:in bit;

out_1:out bit);
end TOGGLE;

architecture behave of TOGGLE is
signal temp:bit;
signal q:bit;
BEGIN
temp <= j XOR q;
DFF(ck,reset,temp,q);
out_1 <= q;
end behave;
```

```
DFF(ck,reset,next_addr,dff_out);

WITH col_end SELECT
mux   <= incr      WHEN '0',
            oct_add_factor WHEN '1';


add<= dff_out + mux;

--architecture  outputs--
out_1 <=    dff_out;

END behave;

CONFIGURATION NOMULT_CON OF U_NOMULT is
FOR behave
END FOR;
END  NOMULT_CON;

-- a toggle flip-flop
use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;

entity JKFF IS
PORT(
 ck : in bit ;
reset : in t_reset ;
j:in bit;

out_1:out bit);
end JKFF;

architecture behave of JKFF is
signal temp:bit;
signal q:bit;
```

EP 0 622 741 A2

```
CONFIGURATION TOGGLE_CON OF TOGGLE  is
FOR behave
END FOR;
END TOGGLE_CON;

--------------------------------------------------------------------------------
--the read and write address generator,input the initial image & block sizes for octave 0 for the y channel--
use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;

entity U_ADDR_GEN IS
PORT(
  ck : in bit ;
  reset : in t_reset ;
  direction : in t_direction ;
  channel : in t_channel ;
  x_p_1 : in  BIT_VECTOR(1 to 10) ;
  x3_p_1 : in BIT_VECTOR(1 to 12) ;
  x7_p_1 : in BIT_VECTOR(1 to 13) ;
  octave_row_length : in  BIT_VECTOR (1 to ysize) ;
  octave_col_length : in BIT_VECTOR (1 to xsize) ;
  octave_reset : in t_reset ;
  octave : in  t_octave ;
  y_done : in bit ;
  uv_done : in bit ;
  octave_finished : in  t_load ;
  base_u,base_v : in  BIT_VECTOR(1 to 19) ;

out_1 : out t_input_mux;            --input data from memory/external

out_2_1 : out t_memory_addr;  -- memory port
out_2_2 : out t_memory_addr;
out_2_3 : out t_load;

out_3_1 : out t_load;   --dwt in control
```

```
out_3_2 : out t_cs;

out_4 : out t_load;          --IDWT data valid
out_5 : out t_load;          --read_valid
out_6 : out t_count_control; --row read

out_7_1 : out t_col;
out_7_2 : out t_count_control);
end U_ADDR_GEN;

--the current octave and when the block finishes the 3 octave transform--

architecture behave OF U_ADDR_GEN IS
COMPONENT U_MEM_CONTROL
PORT(
   ck : in bit ;
   reset : in t_reset ;
   direction : in t_direction ;
   channel : in t_channel ;
   octave : in t_octave ;
   addr_w,addr_r : in    t_memory_addr ;
   zero_hh : in t_load ;

out_1 : out t_input_mux;

out_2_1 : out t_memory_addr;
out_2_2 : out t_memory_addr;
out_2_3 : out t_load;

out_3_1 : out t_load;
out_3_2 : out t_ca);
end COMPONENT;

COMPONENT JKFF
PORT(
ck : in bit ;
```

```
reset : in t_reset ;
j:in bit;

 out_1:out bit);
 end COMPONENT;

COMPONENT U_ROW_COUNT
PORT(
  ck : in bit ;
  reset : in t_reset ;
  octave_cnt_length : in BIT_VECTOR(1 to ysize) ;
  col_carry: in t_count_control;

out_1 : out t_row;
out_2 : out t_count_control);
--count value , and flag for count=0,1,2,row_length-1, row_length
end COMPONENT;

COMPONENT U_COL_COUNT
PORT(
  ck : in bit ;
  reset : in t_reset ;
  octave_cnt_length : in BIT_VECTOR(1 to xsize) ;

out_1 : out t_col;
out_2 : out t_count_control);
--count value , and flag for count=0,1,2,col_length-1, col_length
end COMPONENT;
COMPONENT U_NOMULT
PORT(
  ck : in bit ;
  reset : in t_reset ;
  col_end : in bit ;
  mux_control : in t_mux4 ;
  incr : in t_memory_addr;
  oct_add_factor : in   t_memory_addr ;
  base_u,base_v : in  BIT_VECTOR(1 to 19);
```

```
out_1 : out t_memory_addr);
end COMPONENT; ;

        signal mem_sel:t_mux4;
        signal read_mux:t_mux4;
        signal write_mux:t_mux4;
        signal incr :t_memory_addr;
        signal oct_add_factor :t_memory_addr;
        signal add_2_y :BIT_VECTOR(1 to 13);
        signal add_2_uv  :BIT_VECTOR(1 to 13);
        signal add_2   :BIT_VECTOR(1 to 13);
        signal addr_col_flag    :BIT;
        signal write_latency    :BIT;
        signal read_done    :BIT;
        signal zero_hh    :t_load;
        signal zero_hh_bit    :bit;
        signal read_done_bit    :bit;
        signal start_write_col    :t_load;
        signal read_valid    :t_load;
        signal addr_col_2  :t_count_control;
        signal addr_row_2  :t_count_control;
        signal addr_row_1 :t_row;
        signal addr_col_1 :t_col;
        signal all_done :bit;
        signal all_one :bit;
        signal temp0 :bit;
        signal temp1 :bit;
        signal temp2 :bit;
        signal temp3 :bit;
        signal temp4 :bit;
        signal temp5 :bit;
        signal temp6 :bit;
        signal read_addr:t_memory_addr;
        signal write_addr:t_memory_addr;
        signal mem_control_1:t_input_mux;
```

```
signal mem_control_2_1:t_memory_addr;
signal mem_control_2_2:t_memory_addr;
signal mem_control_2_3:t_load;
signal mem_control_3_1:t_load;
signal mem_control_3_2:t_cs;
BEGIN

WITH octave SELECT
incr    <= 1        WHEN 0,
                    2  WHEN 1,
                    4    WHEN 2,
                    8 WHEN 3;

WITH octave SELECT
add_2_y  <=  B"0000000000001" WHEN 0,
                    B"000" & x_p_1(1 to 8) & B"10"  WHEN 1,
                    B"0" & x3_p_1(1 to 9) & B"100" WHEN 2,
                    x7_p_1(1 to 9) & B"1000" WHEN 3;

WITH octave SELECT
add_2_uv    <=    B"0000000000001"  WHEN 0,
                          B"0000" & x_p_1(1 to 7) & B"10"  WHEN 1,
                          B"00" & x3_p_1(1 to 8) & B"100"    WHEN 2,
                          B"0" & x7_p_1(1 to 8) & B"1000"    WHEN 3;

WITH channel SELECT
add_2       <=  add_2_y WHEN y,
                add_2_uv   WHEN OTHERS;

oct_add_factor  <=  U_TO_I(add_2);

--signals when write must start  delayed 1 tu for use in zero_hh--

--decode to bit--
WITH addr_col_2  SELECT
addr_col_flag  <=    '1'  WHEN    count_carry ,
```

```
                                    '0'   WHEN  OTHERS;

write_latency <=    '1'  WHEN  addr_row_1 = 2 AND addr_col_1 = conv2d_latency-1 ELSE '0';

--read input data done--
read_done <=    '1'  WHEN addr_row_2 = count_carry AND addr_col_flag ='1' ELSE '0';

WITH zero_hh_bit SELECT
zero_hh  <=  write WHEN '1',
                   read  WHEN '0';

WITH read_done_bit SELECT
read_valid <=  write WHEN '1',
                       read  WHEN '0';

DFF(ck,reset,zero_hh,start_write_col);          --1 tu after zero_hh--

read_mux  <=tres  WHEN  y_done='1' AND uv_done='0' AND octave_finished=write AND channel=y ELSE --base_u--
            tres  WHEN  y_done='0' AND uv_done='0' AND octave_finished=write AND channel=u ELSE
            quatro WHEN y_done='0' AND uv_done='1' AND octave_finished=write AND channel=u ELSE
            quatro WHEN y_done='0' AND uv_done='0' AND octave_finished=write AND channel=v ELSE --base v--
            dos   WHEN y_done='0' AND octave_finished=write AND channel=y ELSE                  --base y--
            uno;

write_mux  <=   uno   WHEN zero_hh =write ELSE        -- keep address 0
                      dos   WHEN  channel=    y ELSE                  --base y--
                      tres  WHEN channel=     u ELSE                --base_u--
                      quatro ;                                            --base v--

--note that all the counters have to be reset at the end of an octave, ie on octave_finished--

--the,row&col,counts,for,the,read,address--

col_map: U_COL_COUNT PORT MAP(ck,octave_reset,octave_col_length,addr_col_1,addr_col_2);

row_map: U_ROW_COUNT PORT MAP(ck,octave_reset,octave_row_length,addr_col_2,addr_row_1,addr_row_2);
```

```
all_one <='1';

tog_1:JKFF PORT MAP(ck,octave_reset,write_latency,zero_hh_bit);

tog_2:JKFF PORT MAP(ck,octave_reset,read_done,read_done_bit);

--wEr addresses for sparc mem--

--conv_2d PIPELINE DELAY ON THIS FLAG

DF1(ck,addr_col_flag,temp0);
DF1(ck,temp0,temp1);
DF1(ck,temp1,temp2);
DF1(ck,temp2,temp3);
DF1(ck,temp3,temp4);
DF1(ck,temp4,temp5);
DF1(ck,temp5,temp6);

read_map:U_NOMULT PORT MAP(ck,reset,addr_col_flag,read_mux,incr,oct_add_factor,base_u,base_v,read_addr);

write_map:U_NOMULT PORT MAP(ck,reset,temp6,write_mux,incr,oct_add_factor,base_u,base_v,write_addr);

mem_ctrl_map: U_MEM_CONTROL PORT MAP(ck,reset,direction,channel,octave,write_addr,read_addr,zero_hh,
mem_control_1,mem_control_2_1,mem_control_2_2,mem_control_2_3,mem_control_3_1,mem_control_3_2);
--architecture  outputs--

out_1 <=mem_control_1;

out_2_1<= mem_control_2_1;
out_2_2 <= mem_control_2_2;
out_2_3 <=  mem_control_2_3;

out_3_1 <= mem_control_3_1;
out_3_2 <= mem_control_3_2;

out_4 <=zero_hh;
out_5 <=read_valid;
```

```
out_6 <=addr_row_2;

out_7_1<=addr_col_1;
out_7_2<=addr_col_2;


END;

CONFIGURATION ADDR_GEN_CON OF U_ADDR_GEN is
FOR behave
    FOR ALL:U_NOMULT    USE ENTITY WORK.U_NOMULT(behave);
     END FOR;
    FOR ALL:U_MEM_CONTROL     USE ENTITY WORK.U_MEM_CONTROL(behave);
     END FOR;
    FOR ALL:U_COL_COUNT USE ENTITY WORK.U_COL_COUNT(behave);
     END FOR;
    FOR ALL:U_ROW_COUNT USE ENTITY WORK.U_ROW_COUNT(behave);
     END FOR;
    FOR ALL:JKFF  USE ENTITY WORK.JKFF(behave);
     END FOR;
END FOR;
END  ADDR_GEN_CON;
--the basic 2d convolver for forward transform, rows first then cols for the forward trandform--
-- cols first then rows for the inverse transform
use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;

entity U_CONV_2D  IS
PORT(
  ck : in bit ;
  reset : in t_reset ;
  in_in : in  t_input ;
  direction : in  t_direction ;
  pdel : in  t_scratch_array(1 to 4);
  conv_reset : in       t_reset ;
```

```
    row_flag : in t_count_control ;
    addr_col_read_1 : in t_col ;
    addr_col_read_2 : in t_count_control);

    out_1 : out t_input;
    out_2_1 : out t_scratch_array(1 to 4);
    out_2_2 : out t_col;
    out_2_3 : out t_col;
    out_3 : out t_count_control;
    out_4 : out t_count_control;
    out_5 : out t_count_control);
    end U_CONV_2D;

    --forward direction outputs in row form --
    --   HH HG HH HG  to  to .     --
    --   HG GG HG GG to  to .      --
    --   HH HG HH HG  to  to .     --
    --   HG GG HG GG to  to .      --
    --the inverse convolver returns the raster scan format output data--
    --the convolver automatically returns a 3 octave transform--

    architecture behave OF U_CONV_2D IS

    COMPONENT U_CONV_ROW
    PORT(
       ck : in bit ;
       reset : in t_reset ;
       direction : in t_direction ;
       in_in : in t_input ;
       col_flag : in t_count_control ;

    out_1 : out t_input ) ;
    end COMPONENT;

    COMPONENT U_CONV_COL
    PORT(
```

```
    ck : in bit ;
    reset : in t_reset ;
    direction : in t_direction ;
    in_in : in  t_input ;
    pdel : in t_scratch_array(1 to 4) ;
    row_flag : in t_count_control ;
    col_count_1 : in t_col;
    col_count_2 : in t_count_control;

    out_1 : out t_input;
    out_2 : out t_scratch_array(1 to 4);
    out_3 : out t_col);
    end COMPONENT;

    signal col_count_2:t_count_control;
    signal row_reset:t_reset;
    signal col_reset:t_reset;
    signal temp1:t_reset;
    signal temp2:t_reset;
    signal conv_reset_inv:t_reset;
    signal col_reset_forw:t_reset;
    signal addr_temp1:t_count_control;
    signal addr_temp2:t_count_control;
    signal addr_col_rd_del:t_count_control;
    signal col_flag:t_count_control;
    signal row_temp0:t_count_control;
    signal row_temp1:t_count_control;
    signal row_temp2:t_count_control;
    signal row_temp3:t_count_control;
    signal row_temp4:t_count_control;
    signal row_control:t_count_control;
    signal col_temp0:t_col;
    signal col_temp1:t_col;
    signal col_temp2:t_col;
    signal col_temp3:t_col;
    signal col_count_1:t_col;
```

```
signal addr_temp4:t_count_control;
signal addr_temp3:t_count_control;
signal del_conv_col:t_input;
signal del_conv_row:t_input;
signal del_conv_in:t_input;
signal row_in:t_input;
signal conv_row:t_input;
signal conv_col:t_input;
signal col_in:t_input;
signal del_in:t_input;
signal pdel_out:t_scratch_array(1 to 4);
signal wr_addr:t_col;
BEGIN

--reset must be delayed for row convolver depending on direction of transform
DFl(ck,conv_reset,templ);
DFl(ck,templ,conv_reset_inv);

WITH direction  SELECT
row_reset  <=    conv_reset   WHEN    forward ,
                       conv_reset_inv WHEN inverse ; --pipeline delays in col_conv--

--reset must be delayed for col convolver depending on direction of transform
DFl(ck,conv_reset_inv,temp2);
DFl(ck,temp2,col_reset_forw);

WITH direction  SELECT
col_reset  <=    col_reset_forw  WHEN    forward ,
              conv_reset                 WHEN   inverse ;      --pipeline delays in row_conv--


-- counter flags must be delayed for col convolver depending on pipelining
DFl(ck,addr_col_read_2,addr_templ);
DFl(ck,addr_templ,addr_col_rd_del);

WITH direction SELECT
addr_temp2 <= addr_col_read_2 WHEN forward,
```

```
                addr_col_rd_del WHEN inverse;

DF1(ck,addr_temp2,col_flag);

-- counter flags  must be delayed for row convolver depending on pipelining
DF1(ck,row_flag,row_temp0);
DF1(ck,row_temp0,row_temp1);
DF1(ck,row_temp1,row_temp2);
DF1(ck,row_temp2,row_temp3);

WITH direction SELECT
row_temp4 <= row_temp3 WHEN forward,
             row_flag WHEN inverse;

DF1(ck,row_temp4,row_control);

--pipeline delays for col counter, count value
DF1(ck,addr_col_read_1,col_temp0);
DF1(ck,col_temp0,col_temp1);
DF1(ck,col_temp1,col_temp2);
DF1(ck,col_temp2,col_temp3);
WITH direction   SELECT
col_count_1  <=    col_temp3   WHEN      forward ,
                   addr_col_read_1  WHEN inverse ;

-- similar for carry flag of col counter
DF1(ck,addr_col_rd_del,addr_temp3);
DF1(ck,addr_temp3,addr_temp4);
WITH direction   SELECT
col_count_2  <=    addr_temp4   WHEN      forward ,
                   addr_col_read_2  WHEN inverse ;


--pipeline delays for the convolver values and input value--
DF1(ck,conv_col,del_conv_col);
```

```
DF1(ck,conv_row,del_conv_row);

DF1(ck,in_in,del_in);

WITH direction SELECT
row_in <= del_in WHEN forward,
          del_conv_col WHEN inverse;

row_map: U_CONV_ROW PORT MAP (ck,row_reset,direction,row_in,col_flag,conv_row);

WITH direction SELECT
col_in <= del_conv_row WHEN forward,
          del_in WHEN inverse;

col_map: U_CONV_COL PORT MAP(ck,col_reset,direction,col_in,pdel,row_control,col_count_1,col_count_2,
                                                    conv_col,pdel_out,wr_addr);

--architecture outputs
WITH direction SELECT
out_1 <=    del_conv_col WHEN forward,
                del_conv_row WHEN inverse;

out_2_1<= pdel_out;
out_2_2 <= wr_addr;
out_2_3 <= col_count_1;

out_3 <= row_control;
out_4 <= col_count_2;
out_5 <= col_flag;

end behave;

CONFIGURATION CONV_2D_CON OF U_CONV_2D is
FOR behave
    FOR ALL:U_CONV_COL        USE ENTITY WORK.U_CONV_COL(behave);
     END FOR;
    FOR ALL:U_CONV_ROW        USE ENTITY WORK.U_CONV_ROW(behave);
```

```
    END FOR;
  END FOR;
  END  CONV_2D_CORE;

  -- 1d col convolver, with control --

  use WORK.dwt_types.all;
  use WORK.utils.all;
  use WORK.utils_dwt.all;
  use WORK.dff_package.all;
  -- a b2 line by line resetable counter for the state machines, out->one on rst--
  --carry active on last element of row--
  entity U_COUNTCOL_2  IS
  PORT(
    ck : in bit ;
    reset : in t_reset ;
    carry: in t_count_control;

  out_1 : out t_count_2 )  ;
  end U_COUNTCOL_2;

  architecture behave OF U_COUNTCOL_2 IS
  signal countdel:t_count_2;
  signal countout:t_count_2;
  BEGIN
  PROCESS(ck,reset,carry)
  BEGIN

                IF reset = rst THEN countout <=one;
                ELSIF  countdel=one  AND carry = count_carry THEN countout  <=    two ;
           ELSIF countdel=two AND carry = count_carry  THEN countout  <=    one;
                ELSE null;
                END IF;
  DFl(ck,countout,countdel);
  --architecture  outputs--
  out_1 <=  countdel;
  END PROCESS;
```

```
END behave;
CONFIGURATION COUNTCOL_2_CON OF U_COUNTCOL_2 is
FOR behave
END FOR;
END  COUNTCOL_2_CON;

use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;
entity U_CONV_COL  IS
PORT(
   ck : in bit ;
   reset : in t_reset ;
   direction : in  t_direction ;
   in_in : in  t_input ;
   pdel : in t_scratch_array(1 to 4) ;
   row_flag : in t_count_control ;
   col_count_1 : in t_col;
   col_count_2 : in t_count_control;

out_1 : out t_input;
out_2 : out t_scratch_array(1 to 4);
out_3 : out t_col);
end U_CONV_COL;


architecture behave OF U_CONV_COL IS

--input is data in and, pdel, out from line-delay memories--
-- out is (G,H), and line delay out port.  The row counter is started 1 cycle later to allow for--
--pipeline delay between MULTIPLIER and this unit --

COMPONENT U_COUNTCOL_2
PORT(
   ck : in bit ;
```

```
    reset : in t_reset ;
    carry: in t_count_control;
                %
out_1 : out t_count_2 ) ;
end COMPONENT;

COMPONENT U_ROUND_BITS
PORT(
in_in :in t_scratch;
sel:in t_round;

out_1:out t_input);
end COMPONENT;

COMPONENT U_MULT_ADD
PORT(
    reset : in t_reset ;
    in_in : in t_input ;
    andsel : in  t_and_array(1 to 3) ;
    centermuxsel : in  t_mux_array(1 to 2) ;
    muxsel : in t_mux4_array(1 to 3) ;
    muxandsel : in       t_and_array(1 to 3) ;
    addsel : in t_add_array(1 to 4) ;
    direction : in  t_direction ;
    pdel  : in t_scratch_array(1 to 4) ;

out_1 : out t_scratch_array(1 to 4) );
end COMPONENT;

signal row_control:t_count_control;
signal row_control_del:t_count_control;
signal col_carry:t_count_control;
signal reset_row:t_reset;
signal shift_const:t_round;

signal andsel:t_and_array(1 to 3);
signal muxandsel:t_and_array(1 to 3);
```

```
signal addsel:t_add_array(1 to 4);
signal count:t_count_2;
signal count_delay:t_count_2;
signal centermuxsel:t_mux_array(1 to 2);
signal muxsel:t_mux4_array(1 to 3);
signal mult_add:t_scratch_array(1 to 4);
signal pdel_in:t_scratch_array(1 to 4);
signal pdel_out:t_scratch_array(1 to 4);
signal  pdel1_del:t_scratch;
signal  gh_out:t_scratch;
signal  rb_out:t_scratch;

signal col_count_temp:t_col;
signal wr_addr:t_col;
signal rd_addr:t_col;
signal gh_select:t_mux;
BEGIN
--the code for the convolver--

DFl(ck,reset,reset_row);

--starts row counter 1 cycle after frame start--
--we want the row counter to be 1 cycle behind the col counter for the delay for the--
--pipelined line delay memory--

DFF(ck,reset,col_count_2,col_carry);

--these need to be synchronised to keep the row counter aligned with the data stream--
--also the delay on col_count deglitches the col carryout--

row_control <= row_flag;          --signal for row<=0;1;2;3; last row; etc--

andsel(1) <=  pass WHEN  direction = forward AND count=one ELSE
              zero WHEN  direction = forward AND count=two ELSE
              pass WHEN direction=inverse AND count=two  ELSE
              zero ;
```

```
WITH row_control SELECT
andsel(2)  <=    zero  WHEN  count_0 ,
                 pass   WHEN   OTHERS;

andsel(3) <=  zero WHEN  direction=forward AND row_control = count_0 ELSE
                 pass;

--now the add/sub control for the convolver adders--
WITH count  SELECT
addsel  <=    (add,add,add,subt)   WHEN    one ,
              (add,subt,add,add)   WHEN    two ;

--now the mux control--
centermuxsel <= (left,right) WHEN (direction = forward AND count = one) OR(direction = inverse AND count = two) ELSE
                (right,left);

--the addmuxsel signal--
 muxandsel(1 to 2) <= (pass,andsel(2)) WHEN direction = inverse ELSE
                      (andsel(2),pass) ;
muxandsel(3)    <= zero WHEN direction = inverse AND row_control=count_1 ELSE
                      pass WHEN direction = inverse ELSE
                      andsel(2);

muxsel(1)  <=   dos WHEN direction = inverse AND row_control=count_0 ELSE
                quatro WHEN direction = inverse AND row_control=count_1 ELSE
                tres WHEN direction = inverse AND row_control= count_lm1 ELSE
                dos WHEN direction = inverse ELSE
                uno;
muxsel(2)   <=  tres WHEN direction = inverse AND row_control=count_0 ELSE
                dos WHEN direction = inverse AND row_control=  count_carry ELSE
                uno WHEN direction = inverse ELSE
                dos WHEN direction = forward AND row_control=count_0 ELSE
                tres WHEN direction = forward AND row_control= count_carry ELSE
                uno;
muxsel(3)   <=  uno WHEN direction = inverse ELSE
```

```
                tres WHEN direction = forward AND row_control=count_0 ELSE
                quatro WHEN direction = forward AND row_control= count_carry ELSE
                dos;

COUNT_MAP: U_COUNTCOL_2 PORT MAP(ck,reset_row,col_carry, count);

-- set up the r/w address for the line delay memory
--need 2 delays between wr and rd addr

DFl(ck,col_count_1,col_count_temp);
DFl(ck,col_count_temp,wr_addr);

rd_addr <= col_count_1;

--in the control signals to the mult_add block--
MULT_ADD_MAP: U_MULT_ADD PORT
MAP(reset,in_in,andsel,centermuxsel,muxsel,muxandsel,addsel,direction,pdel_out,mult_add);

--delay to catch the write address
DFl(ck,mult_add(1),pdel_in(1));
DFl(ck,mult_add(2),pdel_in(2));
DFl(ck,mult_add(3),pdel_in(3));
DFl(ck,mult_add(4),pdel_in(4));

--read delay to match MULT delay
DFl(ck,pdel(1),pdel_out(1));
DFl(ck,pdel(2),pdel_out(2));
DFl(ck,pdel(3),pdel_out(3));
DFl(ck,pdel(4),pdel_out(4));


DFl(ck,count,count_del);

gh_select <= right WHEN (direction = inverse AND count_del =one) OR (direction = forward AND count_del = two) ELSE
                left;

DFl(ck,pdel_out(1), pdel1_del);
```

```
gh_out  <=  MUX_2(pdel_in(4),pdel1_del,gh_select);

DF1(ck,row_control,row_control_del);

shift_const <= shift3 WHEN direction = inverse AND (row_control_del=count_1 OR row_control_del=count_2) ELSE
               shift4 WHEN direction = inverse ELSE
               shift5;

RB_MAP: U_ROUND_BITS PORT MAP(gh_out,shift_const,rb_out);
--architecture  outputs--
out_1 <= rb_out;
out_2 <= pdel_in;
out_3 <= wr_addr;

END behave;

CONFIGURATION CONV_COL_CON OF U_CONV_COL is
FOR behave
    FOR ALL:U_ROUND_BITS        USE ENTITY WORK.U_ROUND_BITS(behave);
     END FOR;
    FOR ALL:U_COUNTCOL_2        USE ENTITY WORK.U_COUNTCOL_2(behave);
     END FOR;
    FOR ALL:U_MULT_ADD      USE ENTITY WORK.U_MULT_ADD(behave);
     END FOR;
END FOR;
END  CONV_COL_CON;


-- a \2 line by line resetable counter for the state machines, out->one on rst--
use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;
entity U_COUNT_2  IS
PORT(
  ck : in bit ;
  reset : in t_reset ;
```

```
out_1 : out t_count_2 ) ;
end U_COUNT_2;

architecture behave OF U_COUNT_2 IS
signal countdel:t_count_2;
signal countout:t_count_2;
BEGIN

countout  <=   one  WHEN reset = rst OR countdel= two ELSE
                two ;
DFl(ck,countout,countdel);
--architecture  outputs--
out_1 <=   countdel;

END;

CONFIGURATION COUNT_2_CON OF U_COUNT_2 is
FOR behave
END FOR;
END  COUNT_2_CON;

--the ld convolver, with control and coeff extend--

use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.dff_package.all;
use WORK.utils_dwt.all;


entity U_CONV_ROW  IS
PORT(
   ck : in bit ;
   reset : in t_reset ;
   direction : in t_direction ;
   in_in : in t_input ;
   col_flag : in t_count_control ;
```

```
out_1 : out t_input ) ;
end U_CONV_ROW;

architecture behave OF U_CONV_ROW IS

-- out is (G,H).  The row counter is started 1 cycle later to allow for--
--pipeline delay between MULTIPLIER and this unit --
--the strings give the col & row lengths for this octave--
COMPONENT U_ROUND_BITS
PORT(
in_in :in t_scratch;
sel:in t_round;

out_1:out t_input);
end COMPONENT;

COMPONENT U_COUNT_2
PORT(
   ck : in bit ;
   reset : in t_reset ;

out_1 : out t_count_2 ) ;
end COMPONENT;

COMPONENT U_MULT_ADD
PORT(
    reset : in t_reset ;
   in_in : in t_input ;
   andsel : in  t_and_array(1 to 3) ;
   centermuxsel : in  t_mux_array(1 to 2) ;
   muxsel : in t_mux4_array(1 to 3) ;
   muxandsel : in      t_and_array(1 to 3) ;
   addsel : in t_add_array(1 to 4) ;
   direction : in  t_direction ;
   pdel  : in t_scratch_array(1 to 4) ;
```

```
out_1 : out t_scratch_array(1 to 4) );
end COMPONENT;

 signal reset_col:t_reset;
 signal col_control:t_count_control;
 signal temp:t_and;
 signal andsel:t_and_array(1 to 3);
 signal muxandsel:t_and_array(1 to 3);
 signal addsel:t_add_array(1 to 4);
 signal count:t_count_2;
 signal centermuxsel:t_mux_array(1 to 2);
 signal muxsel:t_mux4_array(1 to 3);
 signal mult_addr:t_scratch_array(1 to 4);
 signal pdel:t_scratch_array(1 to 4);
 signal pdel1_del:t_scratch;
 signal rb_out:t_scratch;
 signal gh_out:t_scratch;
 signal rb_select:t_round;
 signal gh_select :t_mux;

BEGIN

--the code for the convolver--
-- now the state machines to control the convolver--
--First the and gates--

DFl(ck,reset,reset_col);      --starts row counter 1 cycle after frame start--
                              --makes up for the pipeline delay in MULT--
--!!!LATENCY DEOENDENT!!!--

col_control <= col_flag;      --flag when col_count<=0;1;2;col_length;etc--

andsel(1) <=  pass WHEN  direction = forward AND count=one ELSE
                    zero WHEN  direction = forward AND count=two ELSE
                         pass WHEN direction=inverse AND count=two  ELSE
                         zero ;
```

```
WITH col_control SELECT
andsel(2)  <=     zero  WHEN     count_0 ,
                  pass  WHEN  OTHERS;

andsel(3) <=  zero WHEN  direction=forward AND col_control = count_0 ELSE
                         pass;

--now the add/sub control for the convolver adders--
WITH count  SELECT
addsel  <=    (add,add,add,subt)    WHEN      one ,
              (add,subt,add,add)  WHEN    two ;

--now the mux control--
centermuxsel <= (left,right) WHEN (direction = forward AND count = one) OR(direction = inverse AND count = two) ELSE
                         (right,left);

--the addmuxsel signal--
 muxandsel(1 to 2) <= (pass,andsel(2)) WHEN direction = inverse ELSE
                              (andsel(2),pass) ;
muxandsel(3)      <= zero WHEN direction = inverse AND col_control=count_1 ELSE
                             pass WHEN direction = inverse ELSE
                             andsel(2);

muxsel(1)  <=     dos WHEN direction = inverse AND col_control=count_0 ELSE
                        quatro WHEN direction = inverse AND col_control=count_1 ELSE
                        tres WHEN direction = inverse AND col_control= count_lm1 ELSE
                        dos WHEN direction = inverse ELSE
                        uno;
muxsel(2)   <=    tres WHEN direction = inverse AND col_control=count_0 ELSE
                        dos WHEN direction = inverse AND col_control=  count_carry ELSE
                        uno WHEN direction = inverse ELSE
                        dos WHEN direction = forward AND col_control=count_0 ELSE
                        tres WHEN direction = forward AND col_control= count_carry ELSE
                        uno;
muxsel(3)   <=    uno WHEN direction = inverse ELSE
```

EP 0 622 741 A2

```
                    tres WHEN direction = forward AND col_control=count_0 ELSE
                    quatro WHEN direction = forward AND col_control= count_carry ELSE
                    dos;

COUNT_MAP: U_COUNT_2 PORT MAP(ck,reset_col, count);

--join the control signals to the mult_add block--
MULT_ADD_MAP: U_MULT_ADD PORT
MAP(reset,in_in,andsel,centermuxsel,muxsel,muxandsel,addsel,direction,pdel,mult_add);

--pipeline delay for mult-add,unit--

DFl(ck,mult_add(1),pdel(1));
DFl(ck,mult_add(2),pdel(2));
DFl(ck,mult_add(3),pdel(3));
DFl(ck,mult_add(4),pdel(4));

gh_select <= left WHEN (direction = inverse AND count =one) OR (direction = forward AND count =two)  ELSE
                 right;

DFl(ck,pdel(1), pdel1_del);

gh_out  <=  MUX_2(pdel(4),pdel1_del,gh_select);

rb_select <=      shift3 WHEN direction = inverse AND (col_control=count_2 OR col_control=count_3) ELSE
                        shift4 WHEN direction = inverse ELSE
                       shift5;

RB_MAP: U_ROUND_BITS PORT MAP(gh_out,rb_select,rb_out);
--architecture  outputs--
out_1 <= rb_out;

END behave;

CONFIGURATION CONV_ROW_CON OF U_CONV_ROW is
FOR behave
    FOR ALL:U_ROUND_BITS            USE ENTITY WORK.U_ROUND_BITS(behave);
```

```
    END FOR;
    FOR ALL:U_COUNT_2          USE ENTITY WORK.U_COUNT_2(behave);
      END FOR;
    FOR ALL:U_MULT_ADD         USE ENTITY WORK.U_MULT_ADD(behave);
      END FOR;
  END FOR;
  END  CONV_ROW_CON;
  --The basic toggle flip-flop plus and gate for a synchronous counter
  --input t is the toggle ,outputs are q and tc (toggle for next counter
  --stage
  -- reset is synchronous, is active on final count
  use work.DWT_TYPES.all;
  use work.dff_package.all;

  entity BASIC_COUNT is

  PORT(
  ck:in bit ;reset:in t_reset;en:in bit;q:out bit;carry:out bit);
  end BASIC_COUNT;


  architecture behave OF BASIC_COUNT is
  signal dlat:bit;
  signal in_dff:bit;
  signal reset_bit:bit;
  BEGIN
  WITH reset SELECT
  reset_bit <= '0' WHEN rst,
                '1' WHEN no_rst;
  in_dff<=(dlat XOR en) AND reset_bit;
  DFl(ck,in_dff,dlat);
  carry<=dlat AND en;
  q<=dlat;

  END behave;

  configuration basic_count_con of basic_count is
```

```
    FOR behave
    END for;
end basic_count_con;

-- The n-bit macro counter generator, en is the enable, the outputs   #
--are msb(bit 1).....lsb,carry.This is the same order as ELLA strings are stored#

use work.DWT_TYPES.all;

entity COUNT_SYNC is
GENERIC (n:integer);
PORT(
ck:in bit ;
reset:in t_reset;
en:in bit;
q:out bit_vector(1 to n);
carry:out bit);
end COUNT_SYNC;


architecture behave OF COUNT_SYNC is

COMPONENT basic_count
PORT(
ck:in bit ;reset:in t_reset;en:in bit;q:out bit;carry:out bit);
end COMPONENT;

signal enable:bit_vector(1 to n+1);
BEGIN
enable(n+1)<=en;
c1: for i in n downto 1 generate
      bc:   basic_count PORT MAP(ck,reset,enable(i+1),q(i),enable(i));
      end generate;
carry<=enable(1);
end behave;

--configuration for simulation
```

```
CONFIGURATION COUNT_SYNC_CON OF COUNT_SYNC is
FOR behave
     FOR ALL:basic_count USE ENTITY WORK.basic_count(behave);
     END FOR;
END FOR;
END  COUNT_SYNC_CON;

use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;

entity U_COL_COUNT      IS
PORT(
  ck : in bit ;
  reset : in t_reset ;
  octave_cnt_length : in BIT_VECTOR(1 to xsize) ;

out_1 : out t_col;
out_2 : out t_count_control);
--count value , and flag for count=0,1,2,col_length-1, col_length
end U_COL_COUNT;

architecture behave OF U_COL_COUNT IS
COMPONENT COUNT_SYNC
GENERIC (n:integer);
PORT(
ck:in bit ;
reset:in t_reset;
en:in bit;
q:out bit_vector(1 to n);
carry:out bit);
end COMPONENT;
```

```
signal count_control:t_count_control;
signal count_reset:t_reset;
signal count_flag:bit;
signal all_one:bit;
signal count_str:BIT_VECTOR(1 to xsize);
signal count:t_col;

BEGIN
count <= U_TO_I(count_str);

count_control  <=   count_0   WHEN count= 0  ELSE
                              count_1   WHEN    count =     1  ELSE
                              count_2   WHEN    count =     2  ELSE
                              count_3  WHEN     count =     3  ELSE
                              count_lm1 WHEN  count = (U_TO_I(octave_cnt_length) -1) ELSE
                              count_carry WHEN count = U_TO_I(octave_cnt_length) ELSE
                              count_rst;

count_reset <=     rst WHEN reset =rst ELSE
                       rst WHEN count_control = count_carry ELSE
                         no_rst;
all_one <= '1';

count_map:COUNT_SYNC GENERIC MAP(xsize) PORT MAP(ck,count_reset,all_one,count_str,count_flag);--count always enabled

--architecture  outputs--
out_1 <=    count;
out_2 <= count_control;
END behave;

CONFIGURATION COL_COUNT_CON OF U_COL_COUNT is
FOR behave
    FOR ALL:COUNT_SYNC        USE CONFIGURATION WORK.count_sync_con;
     END FOR;
END FOR;
END  COL_COUNT_CON;
```

```
use WORK.dut_types.all;
use WORK.utils.all;
use WORK.utils_dut.all;
use WORK.dff_package.all;

entity U_ROW_COUNT      IS
PORT(
   ck : in bit ;
   reset : in t_reset ;
   octave_cnt_length : in BIT_VECTOR(1 to ysize) ;
   col_carry: in t_count_control;

out_1 : out t_row;
out_2 : out t_count_control);
--count value , and flag for count=0,1,2,row_length-1, row_length
end U_ROW_COUNT;


architecture behave OF U_ROW_COUNT IS
COMPONENT COUNT_SYNC
GENERIC (n:integer);
PORT(
ck:in bit ;
reset:in t_reset;
en:in bit;
q:out bit_vector(1 to n);
carry:out bit);
end COMPONENT;

signal count_control:t_count_control;
signal count_reset:t_reset;
signal count_flag:bit;
signal count_en:bit;
signal count_str:BIT_VECTOR(1 to ysize);
signal count:t_row;
```

```
BEGIN
count <= U_TO_I(count_str);

--count_control  <=   count_0  WHEN reset= rst ELSE
count_control  <=       count_0   WHEN count= 0  ELSE
                              count_1   WHEN    count =     1  ELSE
                              count_2   WHEN    count =     2  ELSE
                              count_3  WHEN     count =     3  ELSE
                              count_lm1 WHEN  count = (U_TO_I(octave_cnt_length) -1) ELSE
                              count_carry WHEN count = U_TO_I(octave_cnt_length) ELSE
                              count_rst;

count_reset <=     rst WHEN reset =rst ELSE
                       rst WHEN count_control = count_carry  AND col_carry = count_carry ELSE
                         no_rst;

count_en <= '1' WHEN col_carry = count_carry ELSE '0';

count_map: COUNT_SYNC GENERIC MAP(ysize) PORT MAP(clk,count_rset,count_en,count_str,count_flag);--count always enabled

--architecture  outputs--
out_1 <=    count;
out_2 <= count_control;
END behave;

CONFIGURATION ROW_COUNT_CON OF U_ROW_COUNT is
FOR behave
    FOR ALL:COUNT_SYNC        USE CONFIGURATION WORK.count_sync_con;
     END FOR;
END FOR;
END ROW_COUNT_CON;
-- create the rising edge function, and a model of a active high DFF.

use work.DWT_TYPES.all;
use work.utils.all;

package dff_package is
```

```
FUNCTION rising_edge (SIGNAL s:bit) return bool;

PROCEDURE DFl(
SIGNAL ck:in bit;SIGNAL d:in integer;SIGNAL q:out integer);

PROCEDURE DFl(
SIGNAL ck:in bit;SIGNAL d:in t_state;SIGNAL q:out t_state);

PROCEDURE DFl(
SIGNAL ck:in bit;SIGNAL d:in t_count_control;SIGNAL q:out t_count_control);

PROCEDURE DFl(
SIGNAL ck:in bit;SIGNAL d:in t_count_2;SIGNAL q:out t_count_2);

PROCEDURE DFl(
SIGNAL ck:in bit;SIGNAL d:in t_reset;SIGNAL q:out t_reset);

PROCEDURE DFl(
SIGNAL ck:in bit;SIGNAL d:in bit;SIGNAL q:out bit);

PROCEDURE DFl(
SIGNAL ck:in bit;SIGNAL d:in t_load;SIGNAL q:out t_load);

PROCEDURE DFl(CONSTANT n:in integer;
SIGNAL ck:in bit;SIGNAL d:in bit_vector;SIGNAL q:out bit_vector);

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in integer;SIGNAL q:out integer);

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_reset;SIGNAL q:out t_reset);

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_count_2;SIGNAL q:out t_count_2);

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_count_control;SIGNAL q:out t_count_control);
```

```
PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in bit;SIGNAL q:out bit);

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_load;SIGNAL q:out t_load);


PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in integer;SIGNAL q:out integer);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_channel;SIGNAL q:out t_channel);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_diff;SIGNAL q:out t_diff);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_mode;SIGNAL q:out t_mode);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in bit;SIGNAL q:out bit);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in BIT_VECTOR;SIGNAL q:out BIT_VECTOR);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_high_low;SIGNAL q:out t_high_low);


PROCEDURE LATCH(
load:in t_load;SIGNAL d:in bit_vector;SIGNAL q:out bit_vector);

PROCEDURE LATCH(
load:in t_load;SIGNAL d:in bit;SIGNAL q:out bit);

end dff_package;
```

```
package body dff_package is

FUNCTION rising_edge (SIGNAL s:bit) return bool is
BEGIN
        IF(s'event) AND (s='1') AND (s'last_value = '0') THEN return t;
          ELSE return f;
        END IF;
END rising_edge;

--THE DF1 flip-flops, NO RESET----------------------
PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in integer;SIGNAL q:out integer) IS
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END DF1;

PROCEDURE DF1(CONSTANT n:integer;
SIGNAL ck:in bit;SIGNAL d:in bit_vector;SIGNAL q:out bit_vector) IS
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END DF1;

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_state;SIGNAL q:out t_state) IS
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END DF1;

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_load;SIGNAL q:out t_load) IS
```

```
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END DF1;


PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_reset;SIGNAL q:out t_reset) IS
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END DF1;

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_count_2;SIGNAL q:out t_count_2) IS
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END DF1;


PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in bit;SIGNAL q:out bit) IS
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END DF1;

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_count_control;SIGNAL q:out t_count_control) IS
BEGIN
IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
```

```
END IF;
END DF1;

--THE DFF flip-flops, with RESET----------------------------
PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in integer;SIGNAL q:out integer) IS
BEGIN
IF reset=rst THEN q<= 0;
ELSIF(rising_edge(ck) = t ) THEN q<=d;
--IF(rising_edge(ck) = t ) THEN IF reset=rst THEN q<= 0;  ELSE  q<=d ;END IF;
ELSE null;
END IF;
END DFF;



PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_reset;SIGNAL q:out t_reset) IS
BEGIN
IF reset=rst THEN q<= rst;
ELSIF(rising_edge(ck) = t ) THEN q<=d;
    ELSE null;
END IF;
END DFF;

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in bit;SIGNAL q:out bit) IS
BEGIN
IF reset=rst THEN q<= '0';
ELSIF(rising_edge(ck) = t ) THEN q<=d;
      ELSE null;
      END IF;
END DFF;

PROCEDURE DFF(
```

```
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_load;SIGNAL q:out t_load) IS
BEGIN
IF reset=rst THEN q<= read;
ELSIF(rising_edge(ck) = t ) THEN q<=d;
    ELSE null;
    END IF;
END DFF;

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_count_2;SIGNAL q:out t_count_2) IS
BEGIN
IF reset=rst THEN q<= one;
ELSIF(rising_edge(ck) = t ) THEN q<=d;
    ELSE null;
    END IF;
END DFF;
PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_count_control;SIGNAL q:out t_count_control) IS
BEGIN
IF reset=rst THEN q<= count_0;
ELSIF(rising_edge(ck) = t ) THEN q<=d;
    ELSE null;
    END IF;
END DFF;


---         THE DFF_INIT FLIP-FLOPS

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in integer;SIGNAL q:out integer) IS
BEGIN
IF reset=rst THEN q<= 0;
ELSIF load=write THEN IF(rising_edge(ck) = t ) THEN q<=d;
            ELSE null;
                        END IF;
END IF;
END DFF_INIT;
```

```
PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in bit;SIGNAL q:out bit) IS
BEGIN
IF reset=rst THEN q<= '0';
ELSIF load=write THEN IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END IF;
END DFF_INIT;


PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_high_low;SIGNAL q:out t_high_low) IS
BEGIN
IF reset=rst THEN q<= low;
ELSIF load=write THEN IF(rising_edge(ck) = t ) THEN q<=d;
        ELSE null;
                END IF;
END IF;
END DFF_INIT;

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_channel;SIGNAL q:out t_channel) IS
BEGIN
IF reset=rst THEN q<= y;
ELSIF load=write THEN IF(rising_edge(ck) = t ) THEN q<=d;
        ELSE null;
                END IF;
END IF;
END DFF_INIT;


PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_mode;SIGNAL q:out t_mode) IS
BEGIN
IF reset=rst THEN q<= still;
ELSIF load=write THEN IF(rising_edge(ck) = t ) THEN q<=d;
```

```
        ELSE null;
                END IF;
END IF;
END DFF_INIT;

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_diff;SIGNAL q:out t_diff) IS
BEGIN
IF reset=rst THEN q<= nodiff;
ELSIF load=write THEN IF(rising_edge(ck) = t ) THEN q<=d;
        ELSE null;
                END IF;
END IF;
END DFF_INIT;

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in BIT_VECTOR;SIGNAL q:out BIT_VECTOR) IS
BEGIN
IF reset=rst THEN q<= ZERO(d'length);
ELSIF load=write THEN IF(rising_edge(ck) = t ) THEN q<=d;
ELSE null;
END IF;
END IF;
END DFF_INIT;


PROCEDURE LATCH(
load:in t_load;SIGNAL d:in bit_vector;SIGNAL q:out bit_vector) IS
BEGIN
IF load=write THEN  q<=d;
ELSE null;
END IF;
END  LATCH;

PROCEDURE LATCH(
load:in t_load;SIGNAL d:in bit;SIGNAL q:out bit) IS
BEGIN
```

```
    IF load=write THEN  q<=d;
    ELSE null;
    END IF;
    END  LATCH;

    end behave;

    END dff_package;
    --the discrete wavelet transform multi-octave/2d transform with edge compensation--
    --when ext & csl are both low latch the setup params from the nubus(active low), as follows--
    --adl(1 to 4)    select function--
    --    0000  load max_octaves,colour,inversebar--
    --    0001  load yimage--
    --    0010  load ximage--
    --jump table values--
    --    0011  load ximage+1--
    --    0100  load 3ximage+3--
    --    0101  load 7ximage+7--
    --    0110  load base u addr--
    --    0111  load base v addr--
    --adl(21 to 22)   max_octaves--
    --adl(23)   luminance/crominancebar active low, 0 is luminance,1 is colour--
    --adl(24)   forward/inversebar active low, 0 is forward, 1 is inverse--
    --adl(5 to 24)    data (bit 24 lsb)--

    use WORK.dwt_types.all;
    use WORK.utils.all;
    use WORK.utils_dwt.all;
    use WORK.dff_package.all;

    entity U_DWT  IS
    PORT(
      ck : in bit ;
      reset : in t_reset ;
      in_in : in  t_input ;
      extwritel,csl: in bit ;
```

```
    adl : in  BIT_VECTOR(1 to 24) ;
    mem : in  t_input ;
    pdel_in : in t_scratch_array(1 to 4));

  out_1 : out t_input;

  out_2 : out t_load_array(1 to 3));

  out_3 : out t_load_array(1 to 3));

  out_4_1 : out t_memory_addr;     -- memory port
  out_4_2 : out t_memory_addr;
  out_4_3 : out t_load;

  out_5_1 : out     t_scratch_array(1 to 4));      --line delay port
  out_5_2 : out     t_col;
  out_5_3 : out     t_col));
  end U_DWT;

  architecture behave OF U_DWT IS
  COMPONENT JKFF
  PORT(
    ck : in bit ;
    reset : in t_reset ;
    j:in bit;

    out_1:out bit));
  end COMPONENT;

  COMPONENT U_CONV_2D
  PORT(
    ck : in bit ;
    reset : in t_reset ;
    in_in : in  t_input ;
    direction : in  t_direction ;
    pdel : in  t_scratch_array(1 to 4));
```

```
    conv_reset : in   t_reset ;
    row_flag : in t_count_control ;
    addr_col_read_1 : in t_col ;
    addr_col_read_2 : in t_count_control);

    out_1 : out t_input;
    out_2_1 : out t_scratch_array(1 to 4);
    out_2_2 : out t_col;
    out_2_3 : out t_col;
    out_3 : out t_count_control;
    out_4 : out t_count_control;
    out_5 : out t_count_control);
    end COMPONENT;

    COMPONENT U_ADDR_GEN
    PORT(
      ck : in bit ;
      reset : in t_reset ;
      direction : in t_direction ;
      channel : in t_channel ;
      x_p_1 : in  BIT_VECTOR(1 to 10) ;
      x3_p_1 : in BIT_VECTOR(1 to 12) ;
      x7_p_1 : in BIT_VECTOR(1 to 13) ;
      octave_row_length : in  BIT_VECTOR (1 to ysize) ;
      octave_col_length : in BIT_VECTOR (1 to xsize) ;
      octave_reset : in t_reset ;
      octave : in  t_octave ;
      y_done : in bit ;
      uv_done : in bit ;
      octave_finished : in  t_load ;
        base_u,base_v : in  BIT_VECTOR(1 to 19) ;

    out_1 : out t_input_mux;          --input data from memory/external

    out_2_1 : out t_memory_addr;      -- memory port
    out_2_2 : out t_memory_addr;
    out_2_3 : out t_load;
```

```
out_3_1 : out t_load;    --dwt in control
out_3_2 : out t_ce;

out_4 : out t_load;      --IDWT data valid
out_5 : out t_load;      --read_valid
out_6 : out t_count_control;    --row read

out_7_1 : out t_col;
out_7_2 : out t_count_control);
end COMPONENT;

signal max_oct:t_octave;
signal max_oct_str:BIT_VECTOR(1 to 2);
signal col_length:BIT_VECTOR(1 to 10);
signal row_length:BIT_VECTOR(1 to 9);
signal channel_factor_st:BIT;
signal channel_factor:t_channel_factor;
signal direction:t_direction;
signal dir:bit;
signal convcol_row:t_count_control;
signal convcol_col:t_count_control;
signal convrow_col:t_count_control;
signal conv_2d_1:t_input;
signal conv_2d_2_1:t_scratch_array(1 to 4);
signal conv_2d_2_2:t_col;
signal conv_2d_2_3:t_col;
signal conv_2d_3:t_count_control;
signal conv_2d_4:t_count_control;
signal conv_2d_5:t_count_control;

signal octave:t_octave;
signal channel:t_channel;
signal octave_finished:t_load;
signal load_octave:t_load;
signal max_oct_1:t_octave;
signal y_done:bit;
```

```
signal uv_done:bit;
signal all_one:bit;

signal octave_sel:t_mux4;
signal octave_row_length:BIT_VECTOR(1 to ysize);

signal conv_reset:t_reset;
signal octave_col_length:BIT_VECTOR(1 to xsize);

signal input_mux:t_input_mux;
signal addr_gen_1:t_input_mux;
signal addr_gen_2_1:t_memory_addr;
signal addr_gen_2_2:t_memory_addr;
signal addr_gen_2_3:t_load;
signal addr_gen_3_1:t_load;
signal addr_gen_3_2:t_ce;
signal addr_gen_4:t_load;
signal addr_gen_5:t_load;
signal addr_gen_6:t_count_control;
signal addr_gen_7_1:t_col;
signal addr_gen_7_2:t_count_control;
signal mem_rw:t_load;
signal mem_r:t_memory_addr;
signal mem_w:t_memory_addr;

signal gl:bit;
signal inverse_out:t_load_array(1 to 3);
signal forward_in:t_load_array(1 to 3);

signal decode_int:natural;
signal decode:BIT_VECTOR(1 to 8);
signal x_p_1:BIT_VECTOR(1 to 10);
signal x3_p_1:BIT_VECTOR(1 to 12);
signal x7_p_1:BIT_VECTOR(1 to 13);
signal base_u:BIT_VECTOR(1 to 19);
signal base_v:BIT_VECTOR(1 to 19);
signal ad14_2:BIT_VECTOR(1 to 3);
```

```
signal load_regs:BIT_VECTOR(1 to 8);
signal conv_in:t_input;
signal row_bit:bit;
signal row_carry_ff:bit;
signal initial_octave:t_octave;
signal initial_channel:t_channel;
signal max_octave_st:BIT_VECTOR(1 to 2);


BEGIN
--must delay the write control to match the data output of conv_2d, ie by conv2d_latency--

--set up the control params--

max_oct     <= U_TO_I(max_octave_st);

WITH channel_factor_st SELECT
channel_factor <=  luminance WHEN '0',
                                color WHEN '1';


WITH dir  SELECT
direction  <=    forward   WHEN  '0' ,
              inverse  WHEN '1';

--set up the octave params--
      convcol_row <=  conv_2d_3;
convcol_col <= conv_2d_4;
convrow_col <= conv_2d_5;
--signals that conv_col, for forward, or conv_row, for inverse, has finished that octave--
--and selects the next octave value and the sub-image sizes--
--row then col, gives write latency
octave_finished <= write WHEN direction = forward AND row_carry_ff = '1'AND convcol_row=count_2 AND convcol_col=count_2
                                        ELSE
--extra row as col then row
                                write WHEN direction = inverse AND row_carry_ff = '1'AND convcol_row=count_2AND convrow_col=count_3
                                        ELSE
```

```
                                read;

--max octaves for u|v--
WITH max_oct  SELECT
max_oct_1  <=   0   WHEN      0|1 ,
                1   WHEN      2 ,
                2  WHEN       3 ;

y_done  <=   '1' WHEN channel = y AND direction = forward AND octave = max_oct
ELSE
                  '1' WHEN channel = y AND direction = inverse AND octave = 0 ELSE
                  '0';

uv_done  <=   '1' WHEN channel = u AND direction = forward AND octave = max_oct_1 ELSE
                    '1' WHEN channel = v AND direction = forward AND octave = max_oct_1 ELSE
              '1' WHEN channel = u AND direction = inverse AND octave = 0 ELSE
              '1' WHEN channel = v AND direction = inverse AND octave = 0 ELSE
              '0';

PROCESS(octave,channel,ck,load_octave)
variable new_oct :t_octave;
variable new_channel :t_channel;
BEGIN
new_oct :=octave;
new_channel := channel;

-- first describe the progression of the octaves for a max_oct decomposition
CASE direction IS
WHEN  forward  =>  CASE octave IS
                              WHEN 0      =>     new_oct :=1;
                              WHEN 1      =>    new_oct :=2 ;
                              WHEN 2|3    =>    new_oct :=3 ;
                              END CASE;

                              IF y_done = '1' OR uv_done = '1' THEN new_oct :=0; ELSE null;
                              END IF;
WHEN  inverse  =>  CASE octave IS
```

```
                WHEN 3  =>   new_oct :=2;
                WHEN 2  =>  new_oct :=1 ;
                WHEN 1|0  =>  new_oct :=0 ;
                           END CASE;

                           CASE channel IS
                           WHEN y      => CASE octave IS
                                      WHEN 0      => CASE channel_factor IS
                                            WHEN  luminance => new_oct:=max_oct;      --watch for colour
                                            WHEN OTHERS       => new_oct:=max_oct_1;
                                            END CASE;
                                      WHEN OTHERS     => null;
                                      END CASE;
                           WHEN u      =>CASE octave IS
                                      WHEN  0     => new_oct:=max_oct_1;
                                      WHEN OTHERS     => null;
                                      END CASE;
                           WHEN v      =>CASE octave IS
                                      WHEN    0   => new_oct:=max_oct;          --move to y
                                      WHEN OTHERS     => null;
                                      END CASE;
                           END CASE;
END CASE;
--the progression of channels is first y then u then v
      CASE  channel_factor    IS
      WHEN  luminance =>       new_channel := y ;
      WHEN  color =>   IF channel = y AND y_done ='1' THEN new_channel := u; ELSE null; END IF;

                                 IF channel = u AND uv_done ='1' THEN new_channel := v;
                                 ELSIF channel =v  AND uv_done ='1' THEN new_channel := y ;
                                 ELSE null;
                                 END IF;
      END CASE;

-- set initial values for octave and channel after reset
      CASE reset IS
      WHEN no_rst => initial_octave<=new_oct;
```

```
    WHEN  rst   => IF direction = forward     THEN initial_octave<=0;
                   ELSIF direction = inverse AND channel=y THEN initial_octave<=max_oct;
                   ELSIF direction = inverse AND (channel=u OR channel=v) THEN  initial_octave<=max_oct_1;
                   END IF;
    END CASE;

  CASE reset IS
  WHEN no_rst => initial_channel<=new_channel;
  WHEN  rst   => initial_channel<=y;
    END CASE;

-- the DFF's for the state machine

DFF_INIT(ck,no_rst,load_octave,initial_octave,octave);
DFF_INIT(ck,no_rst,load_octave,initial_channel,channel);

END PROCESS;


--the block size divides by 2 every octave--
--the u¦v image starts 1/4 size--
octave_sel  <=    uno     WHEN  octave =0 AND channel= y  ELSE
                  dos   WHEN (octave =1  AND channel= y) OR(octave =0 AND (channel= u OR channel =v)) ELSE
                  tres  WHEN (octave =2  AND channel= y) OR(octave =1 AND (channel= u OR channel =v)) ELSE
                  quatro ;

WITH octave_sel SELECT
octave_row_length  <=   row_length  WHEN uno ,
                                    B"0" & row_length(1 to ysize-1)     WHEN dos,
                                    B"00" & row_length(1 to ysize-2)    WHEN tres,
                                    B"000" & row_length(1 to ysize-3)   WHEN quatro;

WITH octave_sel SELECT
octave_col_length  <=   col_length  WHEN uno ,
                                    B"0" & col_length(1 to xsize-1)     WHEN dos,
                                    B"00" & col_length(1 to xsize-2)     WHEN tres,
                                    B"000" & col_length(1 to xsize-3)   WHEN quatro;
```


EP 0 622 741 A2

```
--load next octave, either on system reset, or write finished--
WITH reset  SELECT
load_octave <=  write  WHEN    rst ,
                octave_finished   WHEN  OTHERS;

--reset the convolvers at the end of an octave, ready for the next octave--
--latch pulse to clean it, note 2 reset pulses at frame start--
--FOR SYNC RESET DONT NEED TO LATCH PULSE
--cant glitch as reset&octave_finished dont change at similar times--

conv_reset  <=    rst  WHEN   reset = rst ELSE
                        rst WHEN octave_finished = write ELSE
                        no_rst;

--latch control data off nubus
gl  <=    '1'       WHEN       extwritel = '1' AND csl = '1'
                            ELSE '0';

mem_w <= addr_gen_2_1;  --write addresses--

mem_r <= addr_gen_2_2;  --read addresses--

mem_rw  <=  addr_gen_2_3;

inverse_out <= (write,read,read) WHEN direction=inverse AND octave=0 AND channel=y AND addr_gen_4=write ELSE
               (read,write,read) WHEN direction=inverse AND octave=0 AND channel=u AND addr_gen_4=write ELSE
               (read,read,write) WHEN direction=inverse AND octave=0 AND channel=v AND addr_gen_4=write ELSE
               (read,read,read);

forward_in <= (read,write,write) WHEN direction=forward AND octave=0 AND channel=y AND addr_gen_5=read ELSE
               (write,read,write) WHEN direction=forward AND octave=0 AND channel=u AND addr_gen_5=read ELSE
               (write,write,read) WHEN direction=forward AND octave=0 AND channel=v AND addr_gen_5=read ELSE
               (write,write,write);


--the control section latch values when read from the NUBUS
--a 3x8 decoder, active high outputs selects the load signal for the appropiate register
```

```
adl4_2 <= (adl(2),adl(3),adl(4));

decode_int <=  1 WHEN       adl4_2 = B"000" ELSE
               2 WHEN       adl4_2 = B"001" ELSE
               4 WHEN       adl4_2 = B"010" ELSE
          8 WHEN       adl4_2 = B"011" ELSE
               16 WHEN      adl4_2 = B"100" ELSE
          32 WHEN      adl4_2 = B"101" ELSE
               64 WHEN      adl4_2 = B"110" ELSE
          128 ;

I_TO_S(decode_int, decode);

load_regs <= ALL_SAME(8,gl) AND decode;

DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(8)),adl(21 to 22),max_octave_st);

DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(8)),adl(23),channel_factor_st);
DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(8)),adl(24),dir);

DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(7)),adl(15 to 24),col_length);
DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(6)),adl(16 to 24),row_length);
DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(5)),adl(15 to 24),x_p_1);

DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(4)),adl(13 to 24),x3_p_1);

DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(3)),adl(12 to 24),x7_p_1);

DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(2)),adl(6 to 24),base_u);
DFF_INIT(ck,no_rst,BIT_LOAD(load_regs(1)),adl(6 to 24),base_v);

--sets a flag when row counter moves onto next frame
WITH convcol_row SELECT
row_bit <= '1' WHEN count_carry,
           '0' WHEN OTHERS;

all_one <='1';
```

```
tog_map:JKFF PORT MAP(ck,conv_reset,row_bit,row_carry_ff);

addr_map:U_ADDR_GEN  PORT
MAP(ck,reset,direction,channel,x_p_1,x3_p_1,x7_p_1,octave_row_length,octave_col_length,
                conv_reset,octave,y_done,uv_done,octave_finished,base_u,base_v,
                addr_gen_1, addr_gen_2_1, addr_gen_2_2, addr_gen_2_3, addr_gen_3_1, addr_gen_3_2, addr_gen_4,
                 addr_gen_5, addr_gen_6, addr_gen_7_1, addr_gen_7_2);

WITH addr_gen_1 SELECT
conv_in <=in_in WHEN dwt_in,
                mem WHEN mem_in;

conv_map:U_CONV_2D PORT MAP(ck,reset,conv_in,direction,pdel_in,
conv_reset,addr_gen_6,addr_gen_7_1,addr_gen_7_2,
                        conv_2d_1, conv_2d_2_1, conv_2d_2_2, conv_2d_2_3, conv_2d_3, conv_2d_4, conv_2d_5);

--architecture  outputs--
out_1       <= conv_2d_1;

out_2       <= inverse_out;

out_3       <=forward_in;

out_4_1     <= addr_gen_2_1;
out_4_2     <=addr_gen_2_2;
out_4_3     <=addr_gen_2_3;

out_5_1     <=conv_2d_2_1;
out_5_2     <=conv_2d_2_2;
out_5_3     <=conv_2d_2_3;


END;

CONFIGURATION DWT_CON OF U_DWT is
```

```
FOR behave
    FOR ALL:U_CONV_2D          USE ENTITY WORK.U_CONV_2D(behave);
    END FOR;
FOR ALL:U_ADDR_GEN             USE ENTITY WORK.U_ADDR_GEN(behave);
    END FOR;
FOR ALL:JKFF                   USE ENTITY WORK.JKFF(behave);
    END FOR;

END FOR;
END  DWT_CON;
package dwt_types is
--constant values
constant result_exp:integer:= 14;
--length of result arith
constant    input_exp:integer:= 10;
--length of 1D convolver input/output--
constant    gmax  :integer:= 7;
--maximum shift value for quantisation constant--
constant    result_range :integer:=   2 ** (result_exp-1);
constant    input_range :integer:=   2 ** (input_exp-1);
constant    max_octave:integer:= 3;
--no of octaves:integer:=max_octave +1; can not be less in this example--
constant    no_octave:integer:= max_octave+1;
constant    xsize :integer:=  10;
--no of bits for ximage--
constant    ysize :integer:=  9;
--no of bits for yimage--
constant    ximage:integer:= 319;
--the xdimension -1 of the image; ie no of cols--
constant    yimage:integer:= 239    ;
--the ydimension -1 of the image; ie no of rows--

--int types--
subtype     t_result is integer range  -result_range to result_range-1;
subtype     t_input is integer range  -input_range to input_range-1;
subtype     t_length is integer range 0 to 15;
subtype     t_inp is integer range 0 to 1023;
```

```
subtype     t_blk  is integer range 0 to 3;
subtype     t_sub  is integer range 0 to 3;
subtype     t_col  is integer range 0 to ximage;
subtype     t_row  is integer range 0 to yimage;
subtype     t_carry      is integer range 0 to 1;
subtype     t_quant      is integer range 0 to qmax;
--address for result&dwt memory; ie 1 frame--
subtype     t_memory_addr is integer range 0 to (2 ** max_octave)*( (ximage+1)*(yimage+1)+(ximage+1))-1 ;
subtype     t_octave is integer range 0 to max_octave;

--bit string and boolean types types--
type bool     is  (f,t);
type flag     is (error , ok);
--control signals--
type  t_reset     is (rst,no_rst);
type  t_load      is (write,read);
type  t_load_vec is ARRAY (NATURAL RANGE <>) of t_load;
---r/wbar  control--
TYPE t_mem IS (random,old_mem,new_mem);
type  t_cs          is (no_sel,sel);
type  t_updown    is (down,up);
--up/down counter control--
type  t_diff      is (diff,nodiff);
--diff or not in quantiser--
type t_intra    is (intra,inter);
--convolver mux & and types--
type  t_mux     is (left,right);
type  t_mux3    is (l,c,r);
type  t_mux4    is (uno,dos,tres,quatro);
type  t_add     is (add,subt);
type  t_direction  is (forward,inverse);
--counter types--
type  t_count_2    is (one,two);
--state types--
type  t_mode   is (void,still,stop,send,still,still_send,lpf_send,lpf_still,lpf_stop);
type    t_mode_vec is ARRAY (NATURAL RANGE <>) of t_mode;
type  t_cycle    is (token_cycle,data_cycle,skip_cycle);
```

```
type  t_state   is (up0,up1,zz0,zz1,zz2,zz3,down1);
--type      t_state   is (start,up0,up1,zz0,zz1,zz2,zz3,down1);
type  t_decode  is (load_low,load_high);
type  t_high_low    is (low,high);
type  t_fifo    is (ok_fifo,error_fifo);
--types for the octave control unit--
type  t_channel   is (y,u,v);
type  t_channel_factor   is (luminance,color);
--types for the control of memory ports--
--type      t_sparcport (t_sparc_addr,t_sparc_addr,t_load,t_cs);

-- TYPES FOR DWT CHIP

CONSTANT scratch_exp:Integer:=16;      --length of scratch arith--
CONSTANT conv2d_latency:Integer:=7;      --the 2d convolver latency
constant    scratch_range :Integer:=   2 ** (scratch_exp-1);
subtype t_scratch is Integer range  -scratch_range to scratch_range-1;

type t_scratch_array is array(NATURAL range <>) of t_scratch;

type t_load_array is array(NATURAL range <>) of t_load;

type t_and is (zero,pass);

type t_and_array is array(NATURAL range <>) of t_and;
type t_add_array is array(NATURAL range <>) of t_add;
type t_mux_array is array(NATURAL range <>) of t_mux;
type t_mux4_array is array(NATURAL range <>) of t_mux4;

type t_count_control is (count_0,count_1,count_2,count_3,count_rst,count_carry,count_lm1);
type t_round is (shift3,shift4,shift5);
type t_input_mux is (dwt_in,mem_in);
```

```
FUNCTION U_TO_I(bits: in bit_vector) RETURN natural;
FUNCTION S_TO_I(bits: in bit_vector) RETURN integer;
PROCEDURE I_TO_S(int:in integer; SIGNAL bits:out bit_vector);
end dwt_types;

package body dwt_types is

FUNCTION U_TO_I(bits:bit_vector) RETURN natural IS

variable result: natural:=0;
BEGIN
FOR i IN bits'range LOOP
result:=result*2 + bit'pos(bits(i));
END LOOP;
RETURN result;
END U_TO_I;

FUNCTION S_TO_I(bits:bit_vector) RETURN integer IS

variable temp:bit_vector(bits'range);
variable result: integer:=0;

BEGIN
IF bits(bits'left) = '1' THEN
    temp:=NOT bits;
ELSE
    temp:=bits;
END IF;

FOR i IN bits'range LOOP
result:=result*2 + bit'pos(temp(i));
END LOOP;
IF bits(bits'left) = '1' THEN
    result:=(-result)-1;
END IF;
RETURN result;
END S_TO_I;
```

```
PROCEDURE I_TO_S(int:in integer; SIGNAL bits:out bit_vector) IS
variable result:bit_vector(bits'range);
variable temp: integer;

BEGIN
IF int < 0 THEN
    temp:=-(int+1);
ELSE temp:=int;
END IF;
FOR i IN bits'reverse_range LOOP
result(i):= bit'val(temp rem 2);
temp:=temp/2;
END LOOP;

IF   int<0 THEN
 result:=NOT result;
 result(bits'left):='1';
END IF;

bits<=result;
END I_TO_S;

FUNCTION INT_TO_S(n:natural;SIGNAL int:in integer) RETURN bit_vector IS
variable result:bit_vector(1 to n);
variable temp: integer;

BEGIN
IF int < 0 THEN
   temp:=-(int+1);
ELSE temp:=int;
END IF;
FOR i IN n downto 1 LOOP
result(i):= bit'val(temp rem 2);
temp:=temp/2;
END LOOP;

-- check to see if integer fits in n bits
```

```
ASSERT (temp=0)
REPORT "Int TO BIG FOR n BITS"
SEVERITY FAILURE;

IF  int<0 THEN
  result:=NOT result;
  result(1):='1';
END IF;

RETURN result;
END INT_TO_S;

end dwt_types;
--a model of an ELLA compatible RAM

use work.DWT_TYPES.all;

entity ella_ram is
PORT(
in_data:in t_input;
wr_addr:in t_memory_addr;
rd_addr:in t_memory_addr;
rw:in t_load;

out_data:out t_input);
end ella_ram;

architecture behave of ella_ram is
BEGIN

ram:process
type mem is array(natural range <>) of t_input;

variable memory:mem(0 to 2000);
--variable memory:mem(0 to (2 ** max_octave)*( (ximage+1)*(yimage+1)+(ximage+1))-1);
```

```
BEGIN
wait on rw, wr_addr,rd_addr ;

--IF rw'event AND rw = write THEN memory(wr_addr):=in_data ;
IF rw = write THEN memory(wr_addr):=in_data ;
ELSE null;
END IF;

out_data <= memory(rd_addr);
END PROCESS;
END behave;

CONFIGURATION ELLA_RAM_CON OF ELLA_RAM is
FOR behave
END FOR;
END  ELLA_RAM_CON;

-- ram for scratch memories
use work.DWT_TYPES.all;

 entity scratch_ram is
 PORT(
 in_data:in t_scratch;
 wr_addr:in t_memory_addr;
 rd_addr:in t_memory_addr;
 rw:in t_load;

  out_data:out t_scratch);
  end scratch_ram;

   architecture behave of scratch_ram is
   BEGIN

ram:process
 variable memory:t_scratch_array(0 to 1023);
 --variable memory:mem(0 to (2 ** max_octave)*( (ximage+1)*(yimage+1)+(ximage+1))-1);
```

```
   BEGIN
   wait on rw, wr_addr,rd_addr ;
    --IF rw'event AND rw = write THEN memory(wr_addr):=in_data ;
    IF rw = write THEN memory(wr_addr):=in_data ;
    ELSE null;
    END IF;

       out_data <= memory(rd_addr);
       END PROCESS;
END behave;

 CONFIGURATION SCRATCH_RAM_CON OF SCRATCH_RAM is
 FOR behave
 END FOR;
 END  SCRATCH_RAM_CON;

--the mem control unit for the DWT chip, outputs the memport values for the sparc, and dwt--
--inputs datain from these 2 ports and mux's it to the 2d convolver.--
use WORK.dwt_types.all;
use WORK.utils.all;
use WORK.utils_dwt.all;
use WORK.dff_package.all;

entity U_MEM_CONTROL  IS
PORT(
  ck : in bit ;
  reset : in t_reset ;
  direction : in t_direction ;
  channel : in t_channel ;
  octave : in t_octave ;
  addr_w,addr_r : in    t_memory_addr ;
  zero_hh : in t_load ;

out_1 : out t_input_mux;

out_2_1 : out t_memory_addr;
```

```
out_2_2 : out t_memory_addr;
out_2_3 : out t_load;
out_3_1 : out t_load;
out_3_2 : out t_cs);
end U_MEM_CONTROL;


architecture behave OF U_MEM_CONTROL IS


BEGIN

--the comb. logic for the control of the i/o ports of the chip--

PROCESS(direction,octave,zero_hh)

variable        rw_eparc :t_load;
variable        rw_dwt:t_load;
variable        cs_dwt:t_cs;
variable        input_mux:t_input_mux;
variable        zero_hh_bit:bit;
BEGIN
        rw_eparc := read;
        rw_dwt := read;
        cs_dwt := no_sel;
        input_mux := mem_in;
            zero_hh_bit:='0';

      IF  direction = forward AND octave=0 THEN

                                                          cs_dwt := sel;
                                                              input_mux:= dwt_in;

      ELSIF direction = inverse AND octave=0 AND   zero_hh =write THEN
                                                                rw_dwt := write;
                                                                    cs_dwt:= sel;

      ELSE  null;
```

```
        END IF;

    --rw_sparc = write when ck=1 and zero_hh=write, otherwise = read--
        CASE zero_hh IS
          WHEN write => zero_hh_bit:= '1';
          WHEN OTHERS  =>   zero_hh_bit:= '0';
          END CASE;

          rw_sparc := zero_hh;

    out_1 <= input_mux;

    out_2_3 <= rw_sparc;

    out_3_1 <= rw_dwt;
    out_3_2 <= cs_dwt;

    END PROCESS;
    out_2_1 <=addr_w;
    out_2_2 <=addr_r;

    END;

CONFIGURATION MEM_CONTROL_CON OF U_MEM_CONTROL is
FOR behave
END FOR;
END  MEM_CONTROL_CON;
-- the basic 1d convolver without the control unit--
use work.DWT_TYPES.all;
use work.utils_dwt.all;
entity U_MULT_ADD  IS
PORT(
   reset : in t_reset ;
   in_in : in t_input ;
   andsel : in  t_and_array(1 to 3) ;
   centermuxsel : in  t_mux_array(1 to 2) ) ;
```

```
    muxsel : in t_mux4_array(1 to 3) ;
    muxandsel : in      t_and_array(1 to 3) ;
    addsel : in t_add_array(1 to 4) ;
    direction : in  t_direction ;
    pdel   : in t_scratch_array(1 to 4) ;

  out_1 : out t_scratch_array(1 to 4) );

FUNCTION AND_2 (in1:t_scratch;sel:t_and) RETURN t_scratch IS
BEGIN
CASE sel IS
WHEN pass => RETURN in1;
WHEN zero => RETURN 0;
END CASE;
END;

end U_MULT_ADD;


architecture behave OF U_MULT_ADD IS

COMPONENT U_MULTIPLIER_ST
PORT(
  in_in : in t_input ;

out_1 : out t_scratch_array(1 to 7) )  ;
end COMPONENT;

signal x3:t_scratch;
signal x2:t_scratch;
signal x8:t_scratch;
signal x5:t_scratch;
signal x11:t_scratch;
signal x19:t_scratch;
signal x30:t_scratch;

signal mult:t_scratch_array(1 to 7);
```

```
signal  mux1:t_scratch;
signal  mux2:t_scratch;
signal  mux3:t_scratch;
signal  centermux:t_scratch_array(1 to 2);
signal  and1:t_scratch;
signal  and2:t_scratch;
signal  and3:t_scratch;
signal  and4:t_scratch;
signal  add1init_scratch;
signal  add3init_scratch;
signal  add4init_scratch;
signal  add_out:t_scratch_array(1 to 4);

BEGIN

--the multiplier outputs--
x3 <= mult(1);
x5 <= mult(2);
x11 <= mult(3);
x19 <= mult(4);
x2 <= mult(5);
x8 <= mult(6);
x30 <= mult(7);

--the mux outputs--
        mux1 <= MUX_4(x11,x5,x8,x2,muxsel(1));

        mux2 <= MUX_4(x19,x30,x8,0,muxsel(2));

        mux3 <= MUX_4(x11,x5,x8,x2,muxsel(3));

        centermux <= (MUX_2(pdel(1),pdel(3),centermuxsel(1)),
                MUX_2(pdel(2),pdel(4),centermuxsel(2)) );

-- the AND gates zero the adder inputs every 2nd row--
--the and gate outputs--
        and1 <= AND_2(pdel(2),andsel(1));
```

```
        and2 <= AND_2(pdel(3),andsel(1));
        and3 <= AND_2(centermux(1),andsel(2));
        and4 <= AND_2(centermux(2),andsel(3));

        add1in <= AND_2(mux1,muxandsel(1));
        add3in <= AND_2(mux3,muxandsel(2));
        add4in <= AND_2(x3,muxandsel(3));

    MULT_MAP: U_MULTIPLIER_ST PORT MAP(in_in,mult);

    add_out(1) <= ADD_SUB(and1,add1in,addsel(1));
    add_out(2) <= ADD_SUB(and3,mux2,addsel(2));
    add_out(3) <= ADD_SUB(and4,add3in,addsel(3));
    add_out(4) <= ADD_SUB(and2,add4in,addsel(4));

    --architecture  outputs--
    out_1 <=     add_out;

    END;

    CONFIGURATION MULT_ADD_CON OF U_MULT_ADD is
    FOR behave
        FOR ALL:U_MULTIPLIER_ST   USE ENTITY WORK.U_MULTIPLIER_ST   (behave);
        END FOR;
    END FOR;
    END  MULT_ADD_CON;
    -- the basic multiplier unit of the convolver --
    use WORK.dwt_types.all;
    entity U_MULTIPLIER_ST  IS
    PORT(
      in_in : in t_input ;

    out_1 : out t_scratch_array(1 to 7) )  ;
    end U_MULTIPLIER_ST;

    architecture behave OF U_MULTIPLIER_ST IS
    signal in_s:BIT_VECTOR(1 to input_exp);
```

```
signal x2_st:BIT_VECTOR(1 to input_exp+1);
signal x8_st:BIT_VECTOR(1 to input_exp+3);
signal x4_st:BIT_VECTOR(1 to input_exp+2);
signal x16_st:BIT_VECTOR(1 to input_exp+4);
signal x2:t_scratch:=0;
signal x3:t_scratch:=0;
signal x5:t_scratch:=0;
signal x8:t_scratch:=0;
signal x11:t_scratch:=0;
signal x19:t_scratch:=0;
signal x30:t_scratch:=0;
BEGIN
--the multiplier outputs, fast adder code commented out--
I_TO_S(in_in,in_s);

x2_st <= in_s & B"0";
x2 <= S_TO_I(x2_st);

x8_st <= in_s & B"000";
x8 <= S_TO_I(x8_st);

x3  <=  in_in +  x2;

x4_st <= in_s & B"00";

x5  <=  in_in + S_TO_I(x4_st) ;

x11  <=  x3 + S_TO_I(x8_st);

x16_st <= in_s & B"0000";

x19  <=  x3 + S_TO_I(x16_st);

x30 <= x11 +x19;

--architecture  outputs--
out_1 <=  ( x3,x5,x11,x19,x2,x8,x30);
```

```
END;

CONFIGURATION MULTIPLIER_ST_CON OF U_MULTIPLIER_ST is
FOR behave
END FOR;
END  MULTIPLIER_ST_CON;
use WORK.dwt_types.all;
entity U_ROUND_BITS  IS
PORT(
   in_in : in t_scratch ;
     sel: in t_round ;
out_1 : out t_input);
end U_ROUND_BITS;

architecture behave OF U_ROUND_BITS  IS
signal s1 : BIT_VECTOR(1 to scratch_exp);
signal  shift : BIT_VECTOR(1 to scratch_exp);
signal mab : BIT;
signal cs : BIT;
signal cs_int : t_carry;
signal sel : BIT;
signal sum17 : integer;
signal  sum17_str : BIT_VECTOR(1 to scratch_exp+1);
signal  sum : BIT_VECTOR(1 to scratch_exp);
signal  out_final : BIT_VECTOR(1 to input_exp);
BEGIN

--THIS ASSUMES THAT THE INPUT_EXP = 10111--
--sel chooses a round factor of 3, 4,5--
--the lsb is the right hand of the string,--
--the index 1 of the string is the left hand end, &is the msb--
--so on add ops bit 1 is the carryout--

I_TO_S(in_in,s1);

mab <=  s1(1);
```

```
--needs to be a 16 bit output for the adder--
WITH sel      SELECT
 shift  <=      msb & msb & msb & sl(1 to scratch_exp-3) WHEN shift3,
                msb & msb & msb & msb & sl(1 to (scratch_exp-4)) WHEN shift4,
                msb & msb & msb & msb & msb & sl(1 to scratch_exp-5) WHEN shift5;

--the carry to round; 1/2 value is rounded towards 0--
cs    <=    '0' WHEN sel=shift4 AND  msb='0' AND sl(scratch_exp-3 to scratch_exp)=b"1000" ELSE --round down on
1/2 value
             sl(scratch_exp-3) WHEN sel=shift4 ELSE -- neg. no

            '0'  WHEN sel=shift3 AND  msb='0' AND sl(scratch_exp-2 to scratch_exp) = b"100" ELSE
             sl(scratch_exp-2) WHEN sel=shift3  ELSE

            '0' WHEN sel=shift5 AND  msb='0' AND sl(scratch_exp-4 to scratch_exp)= b"10000" ELSE
             sl(scratch_exp-4);

cs_int <= 1 WHEN cs ='1' ELSE
                0;

sum17 <= cs_int + S_TO_I(shift);

I_TO_S(sum17,sum17_str);

sum   <= sum17_str(2 to scratch_exp+1);

-- 1 signifies the rounded value is in range, 0 that it must be saturated
--these are the 5 msb's from the 13 bit word
sel   <= '1'      WHEN sel=shift3 AND (sum(4 to 7) = b"1111" OR sum(4 to 7) =b"0000") ELSE  --value in range
         '0' WHEN sel=shift3  ELSE

--these are the 3 msb's from the 12 bit word left after taking out the 4 sign extension bits
        '1' WHEN sel=shift4 AND (sum(5 to 7) = b"111" OR sum(5 to 7) =  b"000") ELSE --value in range
         '0' WHEN sel=shift4 ELSE

--these are the 2 msb's from the 11 bit word
         '1' WHEN sel=shift5 AND (sum(6 to 7) = b"11" OR sum(6 to 7) = b"00") ELSE --value in range
```

```
            '0';

out_final <= b"0111111111" WHEN ss1 = '0' AND sum(1) = '0' ELSE -- saturate to 511
             b"1000000001" WHEN ss1 = '0' AND sum(1) = '1' ELSE -- saturate to -511 SEE QUANT FOR REASON
             sum(7 to scratch_exp);

--architecture  outputs--
out_1 <=      S_TO_I(out_final);

END behave;

CONFIGURATION ROUND_BITS_CON OF U_ROUND_BITS is
FOR behave
END FOR;
END  ROUND_BITS_CON;
use work.DWT_TYPES.all;

-- returns a signal with n copies of the zero
package utils is

FUNCTION ZERO ( CONSTANT n:NATURAL) RETURN BIT_VECTOR;

-- returns a signal with n copies of the input bit
FUNCTION ALL_SAME (CONSTANT n:NATURAL; s:bit) RETURN BIT_VECTOR;

-- reverses the bit order
FUNCTION REV (CONSTANT n:natural;in_in:BIT_VECTOR) RETURN BIT_VECTOR;

end utils;

package body utils is
FUNCTION ALL_SAME (CONSTANT n:NATURAL; s:bit) RETURN BIT_VECTOR IS
variable out_b:BIT_VECTOR(1 to n);
BEGIN
for i IN 1 to n LOOP
out_b(i):= s;
```

```
END LOOP;
RETURN out_b;
END ALL_SAME;

FUNCTION ZERO (CONSTANT n:NATURAL) RETURN BIT_VECTOR IS
variable out_b:BIT_VECTOR(1 to n);
BEGIN
for i IN 1 to n LOOP
out_b(i):='0';
END LOOP;
RETURN out_b;
END  ZERO;

FUNCTION REV (CONSTANT n:natural;in_in:BIT_VECTOR) RETURN BIT_VECTOR IS
variable temp:BIT_VECTOR(1 to n);
BEGIN
for i IN 1 to n LOOP
temp(i):=in_in(n-i+ in_in'left);
END LOOP;
RETURN  temp;
END;

END utils;

use work.DWT_TYPES.all;
use work.utils.all;
use work.dff_package.all;

-- returns a signal with n copies of the zero
package utils_dwt is
FUNCTION MUX_4 (in1:t_scratch;in2:t_scratch;in3:t_scratch;in4:t_scratch;sel:t_mux4) RETURN t_scratch;

FUNCTION MUX_2 (in1:t_scratch;in2:t_scratch;sel:t_mux) RETURN t_scratch;

FUNCTION ADD_SUB (in1:t_scratch;in2:t_scratch;addsel:t_add) RETURN t_scratch;

FUNCTION BIT_LOAD(in1:bit) RETURN t_load;
```

```
end utils_dwt;

package body utils_dwt is
FUNCTION MUX_4 (in1:t_scratch;in2:t_scratch;in3:t_scratch;in4:t_scratch;sel:t_mux4) RETURN t_scratch  IS
BEGIN
CASE sel IS
WHEN uno => RETURN in1;
WHEN dos => RETURN in2;
WHEN tres => RETURN in3;
WHEN quatro => RETURN in4;
END CASE;
END;

FUNCTION MUX_2 (in1:t_scratch;in2:t_scratch;sel:t_mux) RETURN t_scratch  IS
BEGIN
CASE sel IS
WHEN left => RETURN in1;
WHEN right => RETURN in2;
END CASE;
END;

FUNCTION ADD_SUB (in1:t_scratch;in2:t_scratch;addsel:t_add) RETURN t_scratch IS
BEGIN
CASE addsel IS
WHEN add => RETURN in1 + in2;
WHEN subt => RETURN in1 - in2;
END CASE;
END;
FUNCTION BIT_LOAD(in1:bit) RETURN t_load IS
BEGIN
CASE in1 IS
WHEN  '1' => RETURN write;
WHEN OTHERS => RETURN read;
END CASE;
END;

END utils_dwt;
```

```
--VHDL Description of Tree Processor/Encoder-Decoder Circuit--
--The state machine to control the address counters/
--only works for 3 octave decomposition in y,2 in u|v/
--these are the addr gens for the x & y adresses of a pixel given the octave/
--subblk no. for each octave.Each x&y address is of the form      /
--    x =    count(5 bits)(blk(3) to blk(octave+1))(s) (octave 0's)    /
--    y =    count(5 bits)(blk(3) to blk(octave+1))(s) (octave 0's)    /
--this makes up the 9 bit address for CIF images                  /
--the blk & s counters are vertical 2 bit  with the lsb in the x coord  /
--and carry out on 3, last counter is both horiz and vertical counter   /
--read_enable enable the block count for the read address, but not the  /
--carry-outs for the mode change, this is done on the write addr cycle  /
--by write_enable, so same address values generated on read & write cycles/

use work.DWT_TYPES.all;
use work.dff_package.all;

entity U_ADDR_GEN IS

port(
   ck : in bit ;
   reset : in  t_reset ;
   new_channel ,channel : in t_channel ;
   load_channel : in t_load ;
   sub_count : in BIT_VECTOR(1 to 2) ;
   col_length : in BIT_VECTOR(1 to xsize) ;
   row_length : in BIT_VECTOR(1 to ysize) ;
   ximage_string : in BIT_VECTOR(1 to xsize) ;
   yimage_string  : in BIT_VECTOR(1 to ysize) ;
   yimage_string_3 : in BIT_VECTOR(1 to 11) ;
   read_enable,write_enable : in bit ;
   new_mode : in  t_mode ;

   out_1 : out t_memory_addr;
   out_2 : out t_octave;
   out_3 : out bit;
```

```
    out_4 : out bit;
    out_5 : out bit;
    out_6 : out t_state );

    end U_ADDR_GEN;

    architecture behave OF  U_ADDR_GEN is

    COMPONENT U_CONTROL_ENABLE
    port(
    ck : in bit ;
    reset : in t_reset ;
    new_channel,channel : in t_channel ;
    c_blk : in BIT_VECTOR(1 to 3) ;
    subband : in BIT_VECTOR(1 to 2) ;
    load_channel : in     t_load ;
    new_mode : in  t_mode ;

     out_1 : out BIT_VECTOR(1 to 3);
     out_2 : out t_octave;
     out_3 : out bit;
     out_4 : out bit;
     out_5 : out t_state) ;

    end COMPONENT;

    COMPONENT COUNTER
    GENERIC (ncount:integer);
    PORT(
    ck:in bit ;
    reset:in t_reset;
    en:in bit;
    x_lpf:in bit_vector(1 to ncount);
    q:out bit_vector(1 to ncount));
    carry:out bit);
    end COMPONENT;
```

```
COMPONENT BLK_SUB_COUNT
PORT(
ck:in bit ;reset:in t_reset;en,cin_en,cout_en:in bit;qout bit_vector(1 to 2);carry:out bit);
end COMPONENT;

signal rv_enable:bit;
signal y_lpf:BIT_VECTOR(1 to ysize-4);
signal x_lpf:BIT_VECTOR(1 to xsize-4);
signal tree_done:bit:='0';
signal lpf_done:bit:='0';
signal lpf_block_done:bit:='0';
signal sub_en:bit;
signal y_en:bit;
signal x_en:bit;
signal blk_en:BIT_VECTOR(1 to 3):=B"000";
signal octave:t_octave:=0;
signal control_4:t_state:=down1;
signal x_count_1:BIT_VECTOR(1 to xsize-4);
signal x_count_2:bit;
signal y_count_1:BIT_VECTOR(1 to ysize-4);
signal y_count_2:bit;
signal blk_count_2:BIT_VECTOR(1 to 3):=B"000";
signal blk_count_1_1:BIT_VECTOR(1 to 2):=B"00";
signal blk_count_1_2:bit;
signal blk_count_2_1:BIT_VECTOR(1 to 2):=B"00";
signal blk_count_2_2:bit;
signal blk_count_3_1:BIT_VECTOR(1 to 2):=B"00";
signal blk_count_3_2:bit;
signal x_msb_out:BIT_VECTOR(1 to xsize-3);
signal x_lsb_out:BIT_VECTOR(1 to 3);
signal y_msb_out:BIT_VECTOR(1 to ysize-3);
signal y_lsb_out:BIT_VECTOR(1 to 3);
signal x_addr:BIT_VECTOR(1 to xsize);
signal y_addr:BIT_VECTOR(1 to ysize);
signal base_rows:BIT_VECTOR(1 to 11);
signal mult_fac:BIT_VECTOR(1 to xsize);
```

```
signal int_addr:integer:=0;
signal temp:integer:=0;
signal address:t_memory_addr;
signal address_x:t_memory_addr;
signal address_y:t_memory_addr;

BEGIN

--size of lpf/2 -1, for y,u|v. 2 because count in pairs of lpf values   #
--lpf same size for all channels!!!#

y_lpf  <=  row_length(1 to ysize-4);
x_lpf  <=  col_length(1 to xsize-4);

x_en<=  '1' WHEN  tree_done='1' OR lpf_block_done= '1' ELSE
       '0';


--clk y_count when all blocks done for subs 1-3, or when final blk done for lpf#
y_en<=  '1' WHEN sub_count = B"00" AND lpf_block_done='1' AND  x_count_2='1' ELSE
        '1' WHEN sub_count /=B"00"  AND  tree_done= '1'AND  x_count_2='1' ELSE
        '0';

--enable the sub band counter#
sub_en<=  '1' WHEN  y_count_2='1' AND y_en='1' ELSE
       '0';

lpf_done <= sub_en WHEN sub_count = B"00" ELSE '0';

WITH channel  SELECT
 x_msb_out<=   x_count_1 & blk_count_3_1(2) WHEN  y,
--always the msb bits#
           B"0" & x_count_1 WHEN  u|v ;

WITH channel  SELECT
y_msb_out<=  y_count_1 & blk_count_3_1(1) WHEN  y,
           B"0" & y_count_1 WHEN  u|v ;
```

```
WITH octave SELECT
--bit2 is lsb;
 x_lsb_out<=   blk_count_2_1(2) & blk_count_1_1(2)& sub_count(2) WHEN 0  ,
               blk_count_2_1(2)& sub_count(2) & '0' WHEN 1,
                sub_count(2) & '0' & '0' WHEN 2,
                b"000" WHEN OTHERS;

WITH octave  SELECT
--bit 1 is msb;
 y_lsb_out<=  blk_count_2_1(1)&blk_count_1_1(1)& sub_count(1) WHEN 0  ,
               blk_count_2_1(1) & sub_count(1) & '0' WHEN 1,
                sub_count(1) & '0' & '0' WHEN 2,
                b"000" WHEN OTHERS;

x_addr  <=  x_msb_out & x_lsb_out;
y_addr  <=  y_msb_out & y_lsb_out;

WITH channel SELECT
base_rows<=b"000000000000" WHEN  y,
        B"0" & yimage_string(1 to ysize)& B"0" WHEN  u,
        yimage_string_3 WHEN  v;

--base address for no of rows for y,u &v memory areas;

WITH channel SELECT
mult_fac<=ximage_string WHEN  y,
        b"0" & ximage_string(1 to xsize-1) WHEN  u|v;

address_x<= U_TO_I(x_addr);

address_y<= U_TO_I(y_addr);

address <= U_TO_I(x_addr) + ( (U_TO_I(y_addr) + U_TO_I(base_rows)) * U_TO_I(mult_fac) ));
```

```
blk_count_2 <= blk_count_1_2 & blk_count_2_2 & blk_count_3_2;
rw_enable <= read_enable OR write_enable;
cnt1: COUNTER GENERIC MAP(xsize-4) PORT MAP(ck,reset,x_en,x_lpf,x_count_1,x_count_2);
cnt2: COUNTER GENERIC MAP(ysize-4) PORT MAP(ck,reset,y_en,y_lpf,y_count_1,y_count_2);
--use new_channel so on channel change control state picks up correct value#
cnt_en:U_CONTROL_ENABLE PORT MAP(ck,reset,new_channel,channel,blk_count_2,
                    sub_count,load_channel,new_mode, blk_en,octave,tree_done,lpf_block_done,control_4);
bsub_1: BLK_SUB_COUNT PORT MAP(ck,reset,blk_en(1),rw_enable,write_enable,blk_count_1_1,blk_count_1_2);
bsub_2: BLK_SUB_COUNT PORT MAP(ck,reset,blk_en(2),rw_enable,write_enable,blk_count_2_1,blk_count_2_2);
bsub_3: BLK_SUB_COUNT PORT MAP(ck,reset,blk_en(3),rw_enable,write_enable,blk_count_3_1,blk_count_3_2);
--procedure outputs#

out_1 <=  address;
out_2 <= octave;
out_3 <=  sub_en;
out_4 <= tree_done;
out_5 <=  lpf_done;
out_6 <= control_4;

end behave;

CONFIGURATION ADDR_GEN_CON OF U_ADDR_GEN is
FOR behave
      FOR ALL : BLK_SUB_COUNT USE CONFIGURATION WORK.BLK_SUB_CON;
   END FOR;
      FOR ALL : COUNTER USE CONFIGURATION WORK.COUNTER_CON;
   END FOR;
     FOR cnt_en : U_CONTROL_ENABLE USE CONFIGURATION WORK.CONTROL_ENABLE_CON;
   END FOR;

END FOR;
END  ADDR_GEN_CON;

--a counter to control the sequencing ofw, token, huffman cycles--
--decide reset is enabled 1 cycle early, and latched to avoid glitches--
--lpf_stop is a is a dummy mode to disable the block writes&huffman data--
--cycles for that block--
```

```
use work.DWT_TYPES.all;
use work.dff_package.all;

entity U_CONTROL_COUNTER  IS
PORT(
   ck : in bit ;
   reset : in t_reset ;
   mode,new_mode : in t_mode ;
   direction  : in t_direction ;

   out_0 : out t_load;
   out_1 : out t_cycle;
   out_2 : out t_reset;
   out_3 : out bit;
   out_4 : out bit;
   out_5 : out t_load;
   out_6 : out t_cs;
   out_7 : out t_load;
   out_8 : out t_cs) ;

   --mode load,cycle,decide reset,read_addr_enable,write_addr_enable,load flags--
   --decode write_addr_enable early and latch to avoid feedback loop with pro_mode--
   --in MODE_CONTROL--
end U_CONTROL_COUNTER;


architecture behave OF U_CONTROL_COUNTER IS
COMPONENT COUNT_SYNC
GENERIC (n:integer);
PORT(
ck:in bit ;
reset:in t_reset;
en:in bit;
q:out bit_vector(1 to n);
carry:out bit);
end COMPONENT;
```

```
signal write_del:bit;
signal write_sig:bit;
signal decide_delay:t_reset;
signal decide_sig:t_reset;
signal count_reset:t_reset;
signal count_len:t_length;
signal count_1:BIT_VECTOR(1 to 4);
signal count_2:bit;
signal always_one:bit:='1';
BEGIN

count_len  <=  U_TO_I( count_1);

control:PROCESS(ck,count_reset,direction,mode,new_mode,count_len)

VARIABLE        cycle : t_cycle;
VARIABLE        decide_reset :  t_reset;
VARIABLE        load_mode : t_load;
VARIABLE        load_flags : t_load;
VARIABLE        cs_new : t_cs;
VARIABLE        cs_old : t_cs;
VARIABLE        rw_old : t_load;
VARIABLE        read_addr_enable : bit;
VARIABLE        write_addr_enable : bit;

BEGIN
                cycle := skip_cycle;
              decide_reset :=  no_rst;
               load_mode := read;
               load_flags := read;
               cs_new := no_sel;
               cs_old := sel;
               rw_old := read;
               read_addr_enable := '0';
               write_addr_enable := '0';

                CASE direction  IS
```

```
WHEN forward =>
    CASE mode IS
        WHEN send|still_send|lpf_send =>
                CASE count_len IS
            WHEN     0 to  3 => read_addr_enable := '1';
                     ca_new:= sel;
            WHEN     4 => cycle := token_cycle;
                     load_flag:= write;
                     write_addr_enable:= '1';
            WHEN     5 to  7 =>    write_addr_enable:= '1';
                     CASE new_mode IS
                     WHEN stop|lpf_stop =>  cycle := skip_cycle;
                                            rw_old:= read;
                                            ca_old:= no_sel;
                     WHEN void =>  cycle := skip_cycle;
                                   rw_old:= write;
                     WHEN  OTHERS  => cycle := data_cycle;
                                      rw_old:= write;
                     END CASE;
            WHEN     8 => decide_reset := rst;
                     CASE new_mode IS
                     WHEN stop|lpf_stop =>  cycle := skip_cycle;
                                            rw_old:= read;
                                            ca_old:= no_sel;
                     WHEN void =>  cycle := skip_cycle;
                                            load_mode:= write;
                                            rw_old:= write;
                     WHEN  OTHERS  => cycle := data_cycle;
                                            load_mode:= write;
                                            rw_old:= write;
                     END CASE;
            WHEN  OTHERS  => null;
            END CASE;
        WHEN still =>    CASE count_len IS
                     WHEN     0 to  3 => read_addr_enable := '1';
                              ca_new:= sel;
```

```
            WHEN    4 =>  cycle := token_cycle;
                          write_addr_enable := '1';
                          load_flage:= write;
                    WHEN  5 to  7 => rw_old := write;
                          write_addr_enable := '1';
                          CASE new_mode  IS
                          WHEN void_still =>  cycle := skip_cycle;
                          WHEN  OTHERS  => cycle := data_cycle;
                          END CASE;

                    WHEN    8 =>  decide_reset := rst;
                          rw_old:= write;
                          load_mode:= write;
                          CASE new_mode  IS
                          WHEN void_still =>  cycle := skip_cycle;
                          WHEN  OTHERS  => cycle := data_cycle;
                          END CASE;

                    WHEN  OTHERS  => null;
                    END CASE;

WHEN lpf_still =>   CASE count_len  IS
                          WHEN 0 to  3 => read_addr_enable := '1';
                                ce_new:= sel;
                          WHEN    4 =>  cycle := token_cycle;
                                write_addr_enable := '1';
                                load_flage:= write;
                          WHEN    5 to  7 => cycle := data_cycle;
                                rw_old:= write;
                                write_addr_enable := '1';
                          WHEN    8 =>  cycle := data_cycle;
                                rw_old:= write;
                                decide_reset:= rst;
                                load_mode:= write;
                          WHEN  OTHERS  => null;
                          END CASE;
```

```
WHEN void =>
    CASE count_len  IS
      WHEN 0 to 3 => read_addr_enable := '1';
                     cs_new:= sel;
      WHEN    4 =>  load_flags := write;
                    cycle:= token_cycle;

--dummy token cycle for mode update--

      WHEN    5 to 7 => write_addr_enable := '1';

--keep counters going--

                      CASE new_mode  IS
                      WHEN stop =>  rw_old := read;
                                    cs_old:= no_sel;
                      WHEN  OTHERS  => rw_old := write;
                      END CASE;
      WHEN    8 =>  decide_reset := rst;
                    CASE new_mode  IS
                    WHEN stop =>  rw_old := read;
                                  cs_old:= no_sel;
                    WHEN  OTHERS  => load_mode := write;
                                     rw_old:= write;
                    END CASE;
    WHEN  OTHERS  => null;
    END CASE;

WHEN void_still =>
    CASE count_len  IS
    WHEN 0 =>  write_addr_enable := '1';

--allow for delay--

    WHEN    1 to  3 => write_addr_enable := '1';
                       rw_old:= write;
    WHEN    4 =>  rw_old := write;
                  load_mode:= write;
                  decide_reset:= rst;
    WHEN  OTHERS  => null;
    END CASE;
  WHEN  OTHERS  => null;
  END CASE;
```

```
                WHEN inverse =>
                    CASE mode  IS
                        WHEN send|still_send|lpf_send =>
                                CASE count_len  IS
                                    WHEN 0 to 3 => read_addr_enable := '1';
                                    WHEN    4 =>  cycle := token_cycle;
                                                  write_addr_enable := '1';
                                                  load_flag:= write;
                                    WHEN    5 to 7 => write_addr_enable := '1';
                                                      CASE new_mode  IS
                                                      WHEN stop|lpf_stop =>  cycle := skip_cycle;
                                                                             rw_old:= read;
                                                                             cs_old:= no_sel;
                                                      WHEN void =>  cycle := skip_cycle;
                                                                    rw_old:= write;
                                                      WHEN  OTHERS  => cycle := data_cycle;
                                                                       rw_old:= write;
                                                      END CASE;
                                    WHEN    8 => decide_reset := rst;
                                                 CASE new_mode  IS
                                                 WHEN stop|lpf_stop =>  cycle := skip_cycle;
                                                                        rw_old:= read;
                                                                        cs_old:= no_sel;
                                                 WHEN void =>  cycle := skip_cycle;
                                                               load_mode:= write;
                                                               rw_old:= write;
                                                 WHEN  OTHERS  => cycle := data_cycle;
                                                                  load_mode:= write;
                                                                  rw_old:= write;
                                                 END CASE;
                                    WHEN  OTHERS  => null;
                                    END CASE;
                        WHEN still =>
                            CASE count_len  IS
                                WHEN      0 => null ;
                                WHEN      1 =>  cycle := token_cycle;
                --skip to allow reset in huffman--
```

```
                write_addr_enable := '1';
            WHEN    2 to  4 => rw_old := write;
                write_addr_enable := '1';
                CASE new_mode  IS
                WHEN void_still => cycle :=skip_cycle;
                WHEN  OTHERS  => cycle := data_cycle;
                END CASE;

        WHEN  5     => rw_old:=write;
                decide_reset:= rst;
                load_mode:= write;
                CASE new_mode  IS
                WHEN void_still =>  cycle :=skip_cycle;
                WHEN  OTHERS  => cycle := data_cycle;
                END CASE;
            WHEN  OTHERS  => null;
            END CASE;
    WHEN lpf_still => CASE count_len  IS
                WHEN
                    0 =>null ;

    --match with previous--

                WHEN     1 =>  write_addr_enable := '1';

    --skip for write enb delay--

                WHEN     2 to  4 => cycle := data_cycle;
                    rw_old:= write;
                    write_addr_enable := '1';
                WHEN     5 =>  cycle := data_cycle;
                    rw_old:= write;
                    decide_reset:= rst;
                    load_mode:= write;
                WHEN  OTHERS  => null;
                END CASE;
            WHEN void =>
            CASE count_len  IS
                WHEN 0 to 3 => read_addr_enable := '1';
                WHEN     4 =>  load_flags := write;
                    cycle:= token_cycle;

    --dummy token cycle for mode update--

                    write_addr_enable := '1';
```

```
WHEN      5 to 7 => write_addr_enable := '1';
                    CASE new_mode  IS
                    WHEN stop =>  rw_old := read;
                                  ce_old:= no_sel;
                    WHEN  OTHERS  => rw_old := write;
                    END CASE;
WHEN      8 =>  decide_reset := rst;
                    CASE new_mode  IS
                    WHEN stop =>  rw_old := read;
                                  ce_old:= no_sel;
                    WHEN  OTHERS  => load_mode := write;
                                  rw_old:= write;
                    END CASE;
WHEN  OTHERS  => null;
END CASE;
  CASE count_len  IS
WHEN        0     =>null ;

WHEN      1 =>  write_addr_enable := '1';

WHEN      2 to  4 => write_addr_enable := '1';
                    rw_old:= write;
WHEN      5 =>  rw_old := write;
                    load_mode:= write;
                    decide_reset:= rst;
WHEN  OTHERS  => null;
END CASE;
```

```
                          WHEN void_still =>

--match with rest--

--dummy as write delayed--




                          WHEN  OTHERS  => null;
                          END CASE;
      END CASE;

write_sig <=write_addr_enable;
decide_sig <= decide_reset;

DFF(ck,reset,write_sig,write_del);
out_0 <= load_mode ;
out_1 <= cycle;
```

```
out_2 <= decide_sig;
out_3 <= read_addr_enable;
out_4 <= write_del;
out_5 <= load_flags;
out_6 <= cs_new;
out_7 <=  rw_old;
out_8 <=  cs_old;

END PROCESS;

WITH reset SELECT
count_reset <= rst WHEN  rst,
                         decide_sig WHEN OTHERS;

control_cnt: count_sync GENERIC MAP(4) PORT MAP(ck,count_reset,always_one,count_1,count_2);

END behave;

CONFIGURATION CONTROL_COUNTER_CON OF U_CONTROL_COUNTER is
FOR behave
    FOR ALL:count_sync  USE ENTITY WORK.count_sync(behave);
     END FOR;
END FOR;
END  CONTROL_COUNTER_CON;
--THE STAte machine to control the address counters
--only works for 3 octave decomposition in y & 2 in u|v

use work.DWT_TYPES.all;
use work.dff_package.all;

entity U_CONTROL_ENABLE is

port(
ck : in bit ;
reset : in t_reset ;
new_channel,channel : in t_channel ;
c_blk : in BIT_VECTOR(1 to 3) ;
```

```
subband : in BIT_VECTOR(1 to 2) ;
load_channel : in      t_load ;
new_mode : in  t_mode ;

out_1 : out BIT_VECTOR(1 to 3);
out_2 : out t_octave;
out_3 : out bit;
out_4 : out bit;
out_5 : out t_state) ;

end U_CONTROL_ENABLE;

architecture behave OF U_CONTROL_ENABLE IS
signal             state:t_state;
signal             new_state_sig:t_state;
BEGIN

state_machine:PROCESS(reset,new_channel,channel,c_blk,subband,load_channel,new_mode,state,new_state_sig)

VARIABLE en_blk:BIT_VECTOR(1 to 3) := B"000";
--enable blk_count#
variable             lpf_block_done:bit := '0';
--enable x_count for LPP#
variable              tree_done:bit := '0';
--enable x_count for other subbands#
variable           reset_state:t_state;
variable              new_state:t_state ;
variable               octave:t_octave := 0;
--current octave#
variable           start_state:t_state;
-- dummy signals for DFl

BEGIN
-- default initial conditions
      en_blk:=b"000";
      lpf_block_done:= '0';
      tree_done:= '0';
```

```
        octave:= 0;
        reset_state:=up0;
        new_state:=state;
        start_state:=up0;
--set up initial state thro mux on reset, on HH stay in zz0 state#
CASE channel  IS
WHEN                    u|v   =>  start_state:=  down1;
WHEN                    y     =>  start_state:=  up0;
END CASE;
CASE reset  IS
WHEN   rst  =>  reset_state:=   start_state;
WHEN  OTHERS      =>  reset_state := state;
END CASE;

CASE  reset_state  IS
WHEN     up0      =>   octave :=2;
                       en_blk(3):= '1';
                       CASE c_blk(3)  IS
                       WHEN  '1'   =>     CASE subband  IS
                                          WHEN     B"00"    =>   lpf_block_done := '1';
--clock x_count for LPF y channel#
                                          WHEN        OTHERS  =>  new_state := up1;
--change state when count done#
                                          END CASE;

                                          CASE new_mode            IS
--in luminance & done with that tree#
                                          WHEN     stop     =>   tree_done := '1';
                                          WHEN        OTHERS  =>  null;
                                          END CASE;
                       WHEN               OTHERS  =>  null;
                       END CASE;
WHEN     up1      =>   octave :=1;
                       en_blk(2):= '1';
                       CASE c_blk(2)  IS
                       WHEN     '1'      =>   new_state := zz0;
```

```
                                    CASE new_mode       IS
--in luminance, terminate branch & move to next branch#
                                    WHEN     stop     =>   new_state := down1;
                                         en_blk(3):= '1';
                                    WHEN        OTHERS  =>  null;
                                    END CASE;
                          WHEN      OTHERS  =>  null;
                          END CASE;
WHEN      zz0       =>  octave :=0;
                          en_blk(1):= '1';
                          CASE c_blk(1)  IS
                          WHEN      '1'       =>   new_state := zz1;
                                    en_blk(2):= '1';
                          WHEN      OTHERS  =>  null;
                          END CASE;
WHEN      zz1       =>  octave :=0;
                          en_blk(1):= '1';
                          CASE c_blk(1)  IS
                          WHEN      '1'       =>   new_state := zz2;
                                    en_blk(2):= '1';
                          WHEN      OTHERS  =>  null;
                          END CASE;
WHEN      zz2       =>  octave :=0;
                          en_blk(1):= '1';
                          CASE c_blk(1)  IS
                          WHEN      '1'       =>   new_state := zz3;
                                    en_blk(2):= '1';
                          WHEN      OTHERS  =>  null;
                          END CASE;
WHEN      zz3       =>  octave :=0;
                          en_blk(1):= '1';
--now decide the next state, on block(1) carry check the other block carries#
                          --nowdecide the next state, on block(1) carry check the other block carries
--
                          CASE c_blk(1)  IS
                          WHEN      '1'       =>   new_state := down1;
                                    en_blk(2):= '1';
```

```
--roll over to 0#
                                    en_blk(3):= '1' ;
--because state zz3 clock 1 pulse#

                          WHEN      OTHERS  =>  null;
                          END CASE;

WHEN      down1     =>    octave :=1;
                          en_blk(2):= '1';
                          CASE c_blk(2)  IS
                          WHEN  '1'   =>      CASE subband  IS
                                              WHEN      B"00"     =>   lpf_block_done := '1' ;
--clock x_count for LPF u|v channel#
                                              WHEN        OTHERS  =>  new_state := zz0 ;
--change state when count done#
                                              END CASE;

                                              CASE new_mode IS
                                              WHEN  stop => CASE channel IS
--stop so finish thisbranch & move on#
                                                        WHEN      u|v       =>   tree_done := '1';
                                                        WHEN      y  =>   en_blk(3) := '1';
                                                                  CASE c_blk(3)       IS
--move to next tree#
                                                                  WHEN      '1'       =>   tree_done := '1';
                                                                  WHEN        OTHERS  =>  new_state := down1;
                                                                  END CASE;

                                                        WHEN      OTHERS  =>  null;
                                                        END CASE;
                                              WHEN        OTHERS  =>  null;
                                              END CASE;
                          WHEN        OTHERS  =>  null;
                          END CASE;

END CASE;
```

```
CASE channel  IS
WHEN  u|v   =>     IF c_blk(1)='1' AND c_blk(2)= '1'  THEN tree_done := '1';
                   ELSE null;
                   END IF;

WHEN  y     =>     IF c_blk(1)='1' AND c_blk(2)='1' AND c_blk(3)= '1'  THEN
                       tree_done := '1';
                  ELSE  null;
                  END IF;
END CASE;

--now change to start state if the sequence has finished#
CASE  tree_done     IS
--in LPF state doesnt change when block done#
WHEN     '1'       =>   new_state := start_state;
WHEN        OTHERS  =>  null;
END CASE;
--on channel change, use starting state for new channel#
CASE    load_channel        IS
--in LPF state doesnt change when block done#
WHEN  write =>     CASE new_channel  IS
                   WHEN  y => new_state:= up0;
                   WHEN  u|v =>  new_state:=down1;
                   END CASE;
WHEN        OTHERS  =>  null;
END CASE;

new_state_sig<=new_state;

out_1 <= en_blk;
out_2 <= octave;
out_3 <= tree_done;
out_4 <= lpf_block_done;
out_5 <=reset_state;

END PROCESS;
```

```
DFl(ck,new_state_sig,state));
END behave;

CONFIGURATION CONTROL_ENABLE_CON OF U_CONTROL_ENABLE is
FOR behave
END FOR;
END  CONTROL_ENABLE_CON;

--The basic toggle flip-flop plus and gate for a synchronous counter
--input t is the toggle ,outputs are q and tc (toggle for next counter)
--stage
-- reset is synchronous, is active on final count
use work.DWT_TYPES.all;
use work.dff_package.all;

entity BASIC_COUNT is

PORT(
ck:in bit ;reset:in t_reset;en:in bit;qout bit;carry:out bit));
end BASIC_COUNT;

architecture behave OF BASIC_COUNT is
signal dlat:bit;
signal in_dff:bit;
signal reset_bit:bit;
BEGIN
WITH reset SELECT
reset_bit <= '0' WHEN rst,
                '1' WHEN no_rst;
in_dff<=(dlat XOR en) AND reset_bit;
DFl(ck,in_dff,dlat));
carry<=dlat AND en;
q<=dlat;

END behave;
```

```
configuration basic_count_con of basic_count is
    FOR behave
    END for;
end basic_count_con;

-- The n-bit macro counter generator, en is the enable, the outputs  #
--are msb(bit 1).....lsb,carry.This is the same order as ELLA strings are stored#

use work.DWT_TYPES.all;

entity COUNT_SYNC is
GENERIC (n:integer);
PORT(
ck:in bit ;
reset:in t_reset;
en:in bit;
q:out bit_vector(1 to n);
carry:out bit);
end COUNT_SYNC;

architecture behave OF COUNT_SYNC is

COMPONENT basic_count
PORT(
ck:in bit ;reset:in t_reset;en:in bit;q:out bit;carry:out bit);
end COMPONENT;

signal enable:bit_vector(1 to n+1);
BEGIN
enable(n+1)<=en;
c1: for i in n downto 1 generate
    bc:   basic_count PORT MAP(ck,reset,enable(i+1),q(i),enable(i));
    end generate;
carry<=enable(1);
end behave;
```

```
--configuration for simulation
CONFIGURATION COUNT_SYNC_CON OF COUNT_SYNC is
FOR behave
     FOR ALL:basic_count USE ENTITY WORK.basic_count(behave);
     END FOR;
END FOR;
END  COUNT_SYNC_CON;

--the basic x/y counter, carry out 1 cycle before final count given by x_lpf/y_lpf#
use work.DWT_TYPES.all;
use work.dff_package.all;

entity COUNTER is
GENERIC (ncount:integer);
PORT(
ck:in bit ;
reset:in t_reset;
en:in bit;
x_lpf:in bit_vector(1 to ncount);
q:out bit_vector(1 to ncount);
carry:out bit);
end COUNTER;

architecture behave OF COUNTER is

 COMPONENT count_sync
GENERIC (n:integer);
 PORT(
 ck:in bit ;reset:in t_reset;en:in bit;q:out bit_vector(1 to ncount);carry:out bit);
 end COMPONENT;

   signal cnt_reset:t_reset;
   signal final_count:bit;
   signal final_cnt_d:bit;
   signal q_sync:bit_vector(1 to ncount);
   signal carry_sync:bit;
BEGIN
```

```
   cnt_sy: count_sync GENERIC MAP(ncount) PORT MAP(ck,cnt_reset,en,q_sync,carry_sync);

q<=q_sync;
carry<=final_count;

final_count <= '1' WHEN q_sync=x_lpf AND en = '1' ELSE '0';

cnt_reset <= rst WHEN reset=rst  ELSE
                  rst WHEN final_count = '1' ELSE
                  no_rst;
END behave;

CONFIGURATION COUNTER_CON OF COUNTER is
FOR behave
FOR ALL:count_sync USE CONFIGURATION WORK.count_sync_con;
END FOR;
END FOR;
END  COUNTER_CON;

--the blk, or sub-band counters, carry out on 3, cout_en enables the carry out, & cin_en AND en  enables the
count# use work.DWT_TYPES.all;

entity BLK_SUB_COUNT is

PORT(
ck:in bit ;reset:in t_reset;en,cin_en,cout_en:in bit;q:out bit_vector(1 to 2);carry:out bit);
end BLK_SUB_COUNT;


architecture behave OF BLK_SUB_COUNT is
COMPONENT count_sync
GENERIC (n:integer);
  PORT(
        ck:in bit ;reset:in t_reset;en:in bit;q:out bit_vector(1 to 2);carry:out bit);
end COMPONENT;

signal q_sync:bit_vector(1 to 2);
```

```
signal carry_sync:bit;
signal enable:bit;

BEGIN

enable <= en AND cin_en;
q<=q_sync;
carry<= '1' WHEN q_sync = b"11" AND cout_en = '1' ELSE '0';

        b_cnt: count_sync GENERIC MAP(2) PORT MAP(ck,reset,enable,q_sync,carry_sync);
END behave;

CONFIGURATION BLK_SUB_CON OF BLK_SUB_COUNT is
  FOR behave
          FOR b_cnt : count_sync USE CONFIGURATION WORK.count_sync_con;

          END FOR;
  END FOR;
 END BLK_SUB_CON;
--the L1 norm comparison constantes flag values--
--adding 4 absolute data values so result can grow by 2 bits--
--5 cycle sequence, a reset cycle with no data input, followed--
--by 4 data cycles--

use work.DWT_TYPES.all;
use work.dff_package.all;
use work.utils.all;

entity U_L1NORM  IS
GENERIC(n:integer);
PORT(
  ck : in bit ;
  reset : in  t_reset ;
  in_a : in  BIT_VECTOR(1 to n) ;

out_1 : out BIT_VECTOR(1 to n+2) );
```

```
end U_LINORM;

architecture behave OF U_LINORM IS

  signal msb:BIT_VECTOR(1 to n) ;
  signal add_in1:BIT_VECTOR(1 to n) ;
  signal rst_mux:BIT_VECTOR(1 to n+4) ;
  signal in2:BIT_VECTOR(1 to n+4) ;
  signal add_out:BIT_VECTOR(1 to n+5) ;
  signal carry:t_carry;
  signal adder :integer;

BEGIN

  msb  <=  ALL_SAME(n,in_s(1));
  add_in1  <=  (in_s XOR msb);

  WITH reset  SELECT
  rst_mux  <=    ZERO(n+4)  WHEN      rst ,
                        add_out(2 to n+5) WHEN  OTHERS;

  --carryin bit to adder
  carry <= 1  WHEN in_s(1)='1' ELSE
             0;

  adder  <= S_TO_I(add_in1) + S_TO_I(in2) + carry;

  I_TO_S(adder,add_out);

  DFl(n+4,ck,rst_mux,in2);
  --procedure outputs--

  out_1 <=  in2(3 to n+4);

  END;
```

```
CONFIGURATION U_L1NORM_CON OF U_L1NORM is
FOR behave
END FOR;
END  U_L1NORM_CON;
--the block to decide if all its inputs are all 0--

use work.DWT_TYPES.all;
use work.dff_package.all;

entity U_ALL_ZERO  IS
PORT(
  ck : in bit ;
  reset : in  t_reset ;
  in_in : in  t_input ;

out_1 : out bit );

end U_ALL_ZERO;


architecture behave OF U_ALL_ZERO IS

signal out_b:bit;
signal in_eq_0:bit;
signal all_eq_0:bit;
BEGIN

in_eq_0  <=  '1' WHEN in_in = 0 ELSE      --in =0--
                        '0';

--1 if reset high; & OR with previous flag--

all_eq_0  <= in_eq_0  WHEN  reset =  rst ELSE
                '0'  WHEN  out_b='0'  ELSE
                in_eq_0;

DF1(ck,all_eq_0,out_b);
```

```
--procedure outputs--

out_1 <=  out_bit;

END;

CONFIGURATION U_ALL_ZERO_CON OF U_ALL_ZERO is
FOR behave
END FOR;
END  U_ALL_ZERO_CON;

use work.DWT_TYPES.all;
use work.dff_package.all;
use work.utils.all;

entity U_ABS_NORM  IS
GENERIC(n:positive);
PORT(
   ck : in bit ;
   reset : in  t_reset ;
   qshift : in BIT_VECTOR(1 to result_exp-2) ;
   in_in  : in  t_input:=0;

out_1 : out BIT_VECTOR(1 to n+2);
out_2 : out bit);

end U_ABS_NORM;


architecture behave OF U_ABS_NORM IS

signal adder_str:BIT_VECTOR(1 to n+5);
signal rst_mux:BIT_VECTOR(1 to n+4);
signal in2:BIT_VECTOR(1 to n+4);
signal add_s:BIT_VECTOR(1 to n+4);
signal adder:integer:=0;
signal abs_in:integer:=0;
```

```
signal in_small:bit;
signal all_small:bit;
signal out_b:bit;
BEGIN

abs_in  <=  abs(in_in);

adder  <=  abs_in + S_TO_I(in2);

I_TO_S(adder,adder_str);

add_s  <= adder_str(2 to (n+5));

WITH reset  SELECT
rst_mux  <=    ZERO(n+4)  WHEN rst ,
           add_s WHEN  OTHERS;

in_small  <= '1' WHEN  abs_in <= U_TO_I(qshift) ELSE
                    '0';

--1 if reset high, & OR with previous flag--

all_small  <=  '1'  WHEN  reset= rst ELSE
                       '0'  WHEN   in_small = '0' ELSE
                       out_b;

DFl(n+4,ck,rst_mux,in2);
DFl(ck,all_small,out_b);
--procedure outputs--

out_1 <=  in2(3 to n+4);
out_2 <= out_b;

END;

CONFIGURATION U_ABS_NORM_CON OF U_ABS_NORM is
FOR behave
```

```
END FOR;
END  U_ABS_NORM_CON;

--the decide fn block--
use work.DWT_TYPES.all;
use work.dff_package.all;
use work.utils.all;

entity U_DECIDE  IS
PORT(
   ck : in bit ;
   reset : in t_reset ;
   q_int : in t_result ;
   nw,old : in t_input ;
      threshold,comparison : in  t_result ;
   octs : in t_octave ;
   load_flags  : in t_load ;

out_1 : out BIT_VECTOR(1 to 7) );

end U_DECIDE;


architecture behave OF U_DECIDE IS

COMPONENT U_LINORM
 GENERIC(n:integer);
 PORT(
 ck : in bit ;
 reset : in  t_reset ;
 in_s : in  BIT_VECTOR(1 to n) ;

out_1 : out BIT_VECTOR(1 to n+2) );
end COMPONENT;

COMPONENT U_ABS_NORM
 GENERIC(n:positive);
```

```
PORT(
ck : in bit ;
reset : in  t_reset ;
qshift : in BIT_VECTOR(1 to result_exp-2) ;
 in_in  : in  t_input;

 out_1 : out BIT_VECTOR(1 to n+2);
 out_2 : out bit);
 end COMPONENT;

--nzflag,origin,noflag,ozflag,motion,pro_new_z,pro_no_z--

 signal nz_plus_oz:BIT_VECTOR(1 to input_exp+3);
 signal shift_add:BIT_VECTOR(1 to input_exp+3);
 signal nw_str:BIT_VECTOR(1 to input_exp);
 signal old_str:BIT_VECTOR(1 to input_exp);
 signal q_int_str :BIT_VECTOR(1 to result_exp);
 signal n_o_str:BIT_VECTOR(1 to input_exp+1);
 signal nz_1:BIT_VECTOR(1 to input_exp+2);
 signal oz_1:BIT_VECTOR(1 to input_exp+2);
 signal no_1:BIT_VECTOR(1 to input_exp+3);
 signal qshift:BIT_VECTOR(1 to result_exp-2);
 signal flags:BIT_VECTOR(1 to 7);
 signal decide_flags:BIT_VECTOR(1 to 7);
 signal n_o: integer;
 signal nz: natural:=0;
 signal oz: natural:=0;
 signal no: natural:=0;
 signal nzflag: bit;
 signal ozflag: bit;
 signal noflag: bit;
 signal origin: bit;
 signal motion: bit;
 signal new_z: bit;
 signal no_z: bit;
 signal nz_2: bit;
 signal no_2: bit;
```

```
signal octs_del: t_octave;
BEGIN

      I_TO_S(q_int,q_int_str);

      qshift <= q_int_str(1 to result_exp-2);
--divide by 4 as test is on coeff values not block values--

      n_o  <= nw - old;                                    --new-old;use from quant--

-- convert to string for LINORM
      I_TO_S(n_o,n_o_str);
      I_TO_S(nw,nw_str);
      I_TO_S(old,old_str);

--convert to unsigned integer
      nz <= U_TO_I(nz_1);
      oz <= U_TO_I(oz_1);
      no <= U_TO_I(no_1);

      nzflag  <=  '1' WHEN nz <= threshold ELSE          --delay tests for pipelined data--
                  '0';
      noflag  <=  '1' WHEN no <= comparison ELSE
                  '0';
      ozflag  <=  '1' WHEN oz =  0 ELSE
                  '0';
      origin  <=  '1' WHEN nz <= no ELSE
                  '0';
      I_TO_S(nz + oz,nz_plus_oz);

      new_z  <=  nz_2;

      no_z  <=  no_2;

--delay octs to match pipelin delay--

DF1(ck,octs,octs_del);
```


EP 0 622 741 A2

```
--keep 13 bits here to match no; keep msb's--
 --delay octs to match pipelin delay--
WITH octs_del SELECT

shift_add <=  nz_plus_oz(1 to input_exp+3) WHEN 0,
                  B"0"& nz_plus_oz(1 to input_exp+2) WHEN 1,
                  B"00"& nz_plus_oz(1 to input_exp+1) WHEN 2,
                  B"000"& nz_plus_oz(1 to input_exp)  WHEN 3;

motion  <=  '1' WHEN U_TO_I(shift_add) <= no ELSE
                  '0';

decide_flags <= nzflag&origin&noflag&ozflag&motion&new_z&no_z;

abs_1: U_ABS_NORM GENERIC MAP(input_exp) PORT MAP(ck,reset,qshift,nw,nz_1,nz_2);

LATCH(7,load_flags,decide_flags,flags);

abs_2: U_ABS_NORM GENERIC MAP(input_exp+1) PORT MAP(ck,reset,qshift,n_o,no_1,no_2);

l1: U_L1NORM GENERIC MAP(input_exp) PORT MAP(ck,reset,old_str,oz_1);

--procedure outputs--

out_1 <=  flags   ;

END;

CONFIGURATION U_DECIDE_CON OF U_DECIDE is
FOR behave
    FOR ALL: U_ABS_NORM USE ENTITY WORK.U_ABS_NORM(behave);
     END FOR;
    FOR ALL: U_L1NORM   USE ENTITY WORK.U_L1NORM(behave);
     END FOR;
END FOR;
END  U_DECIDE_CON;
-- create the rising edge function, and a model of a active high DFF.
```

```
use work.DWT_TYPES.all;
use work.utils.all;

package dff_package is

FUNCTION rising_edge (SIGNAL s:bit) return bool;

PROCEDURE DF1_LOAD(
SIGNAL ck:in bit;load:in t_load;SIGNAL d:in BIT_VECTOR;SIGNAL q:out BIT_VECTOR);

PROCEDURE DF1_LOAD(
SIGNAL ck:in bit;load:in t_load;SIGNAL d:in t_high_low;SIGNAL q:out t_high_low);

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in integer;SIGNAL q:out integer);

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_direction;SIGNAL q:out t_direction);

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_state;SIGNAL q:out t_state);

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in t_reset;SIGNAL q:out t_reset);

PROCEDURE DF1(
SIGNAL ck:in bit;SIGNAL d:in bit;SIGNAL q:out bit);

PROCEDURE DF1(CONSTANT n:in integer;
SIGNAL ck:in bit;SIGNAL d:in bit_vector;SIGNAL q:out bit_vector);

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in integer;SIGNAL q:out integer);
```

```
PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_reset;SIGNAL q:out t_reset);
PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in bit;SIGNAL q:out bit);

PROCEDURE DFF(
SIGNAL ck:in bit;reset:in t_reset;SIGNAL d:in t_load;SIGNAL q:out t_load);


PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in integer;SIGNAL q:out integer);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_channel;SIGNAL q:out t_channel);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_diff;SIGNAL q:out t_diff);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_mode;SIGNAL q:out t_mode);

PROCEDURE DFF_INIT(CONSTANT n:natural;
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in BIT_VECTOR;SIGNAL q:out BIT_VECTOR);

PROCEDURE DFF_INIT(
SIGNAL ck:in bit;reset:in t_reset;load:in t_load;SIGNAL d:in t_high_low;SIGNAL q:out t_high_low);


PROCEDURE LATCH(CONSTANT n:in integer;
load:in t_load;SIGNAL d:in bit_vector;SIGNAL q:out bit_vector);

end dff_package;


package body dff_package is
```